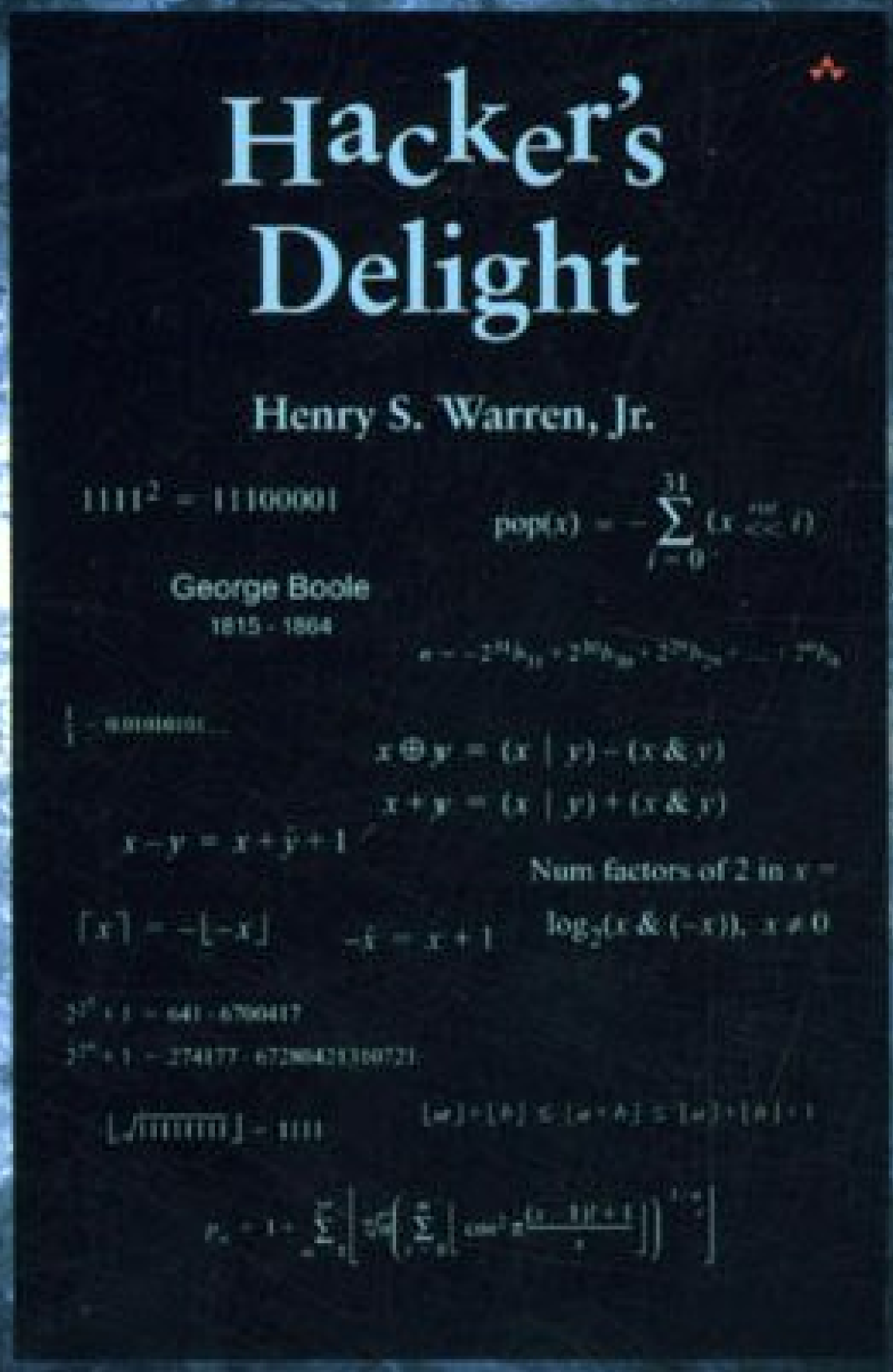
计 算 机 科 学 丛 书

高效程序的奥秘

(美) Henry S. Warren, Jr. 著 冯速 译



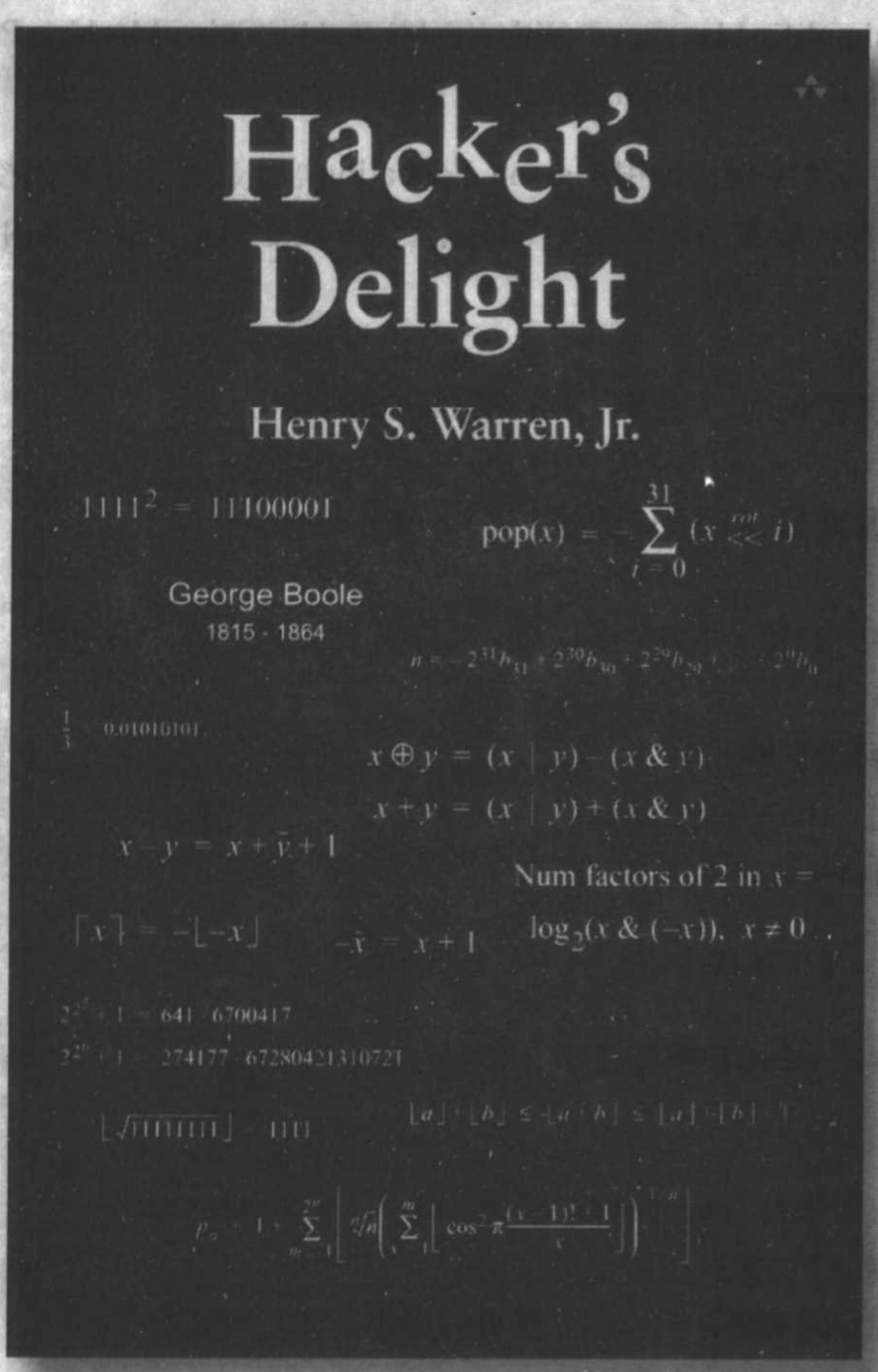
Hacker's Delight

机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

高效程序的奥秘

(美) Henry S. Warren, Jr. 著　冯 速 译



Hacker's Delight

机 械 工 业 出 版 社
China Machine Press

本书适合程序库、编译器开发者及追求优美程序设计的人员阅读，适合用作计算机专业高年级学生及研究生的参考用书。

本书直观明了地讲述了计算机算术的更深层次的、更隐秘的技术，汇集了各种编程小技巧，包括常见任务的小算法、2的幂边界和边界检测、位和字节的重排列、整数除法和常量除法、针对整数的基本函数、Gray码、Hilbert空间填充曲线、素数公式等。

本书版权登记号：图字：01-2003-0998

图书在版编目（CIP）数据

高效程序的奥秘 /（美）沃瑞恩（Warren, H. S.）著；冯速译. -北京：机械工业出版社，2004.5

（计算机科学丛书）

书名原文：Hacker's Delight

ISBN 7-111-14111-3

Ⅰ.高 … Ⅱ.① 沃 … ② 冯 … Ⅲ.软件开发 Ⅳ.TP311.52

中国版本图书馆CIP数据核字（2004）第018070号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：刘立卿

北京中加印刷有限公司印刷 · 新华书店北京发行所发行

2004年5月第1版第1次印刷

787mm × 1092mm 1/16 · 16 印张

印数：0 001 - 4 000 册

定价：28.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及收藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入了一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经二十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

秘　书　组

武卫东　　温莉芳　　刘　江　　杨海玲

译 者 序

本书也许会激起某些读者的怀旧情感，特别是那些经历过或曾向往在早期的Z80等计算机上用汇编语言或Basic语言编程的人们；那些竭尽全力、苦思冥想、仅仅是为了能够写出更简短、更高效程序的人们；那些以编写一行高效Basic程序而自我陶醉的人们。

IBM资深程序员、蓝基因千万亿浮点计算机项目参与者Henry S. Warren，Jr.给所有程序员带来一本他们到处寻觅、实实在在、具有现实意义的程序设计技巧指南。本书是一本有特色的算法参考书，它不像现今多数算法书那样讨论大型的程序设计技术和系统的程序设计方法，而是向读者展示了计算机代码与指令，以及指令间的令人惊叹的内在关联，并通过这样的内在关联向读者揭示计算机运行的某些奥秘，揭示通过这样的关联更有效地实现基本操作的精湛技巧，以及这些技巧在优化编译器、图形学、密码学乃至数学中的应用。

本书适用于那些想要编写及欣赏巧妙、高效代码的读者，特别适合于希望把计算机软件和硬件有机地结合在一起的读者。本书值得每一位认真的计算机硬件设计者，每一位希望编写高效、优雅的嵌入式程序、编译器及优化编译器的设计者，以及每一位希望提高程序设计技艺的读者仔细斟酌和品味。

为了方便我国读者，我们尽量以通俗的语言翻译原文，并对难以理解的部分适当做了补充说明。翻译过程中译者得到机械工业出版社华章分社的大力支持。由于本书覆盖的内容较广、翻译时间仓促、水平有限，难免有疏漏与错误之处，请读者不吝指教。

冯 速

2003年12月

于北京师范大学

序

大约30年前，当我第一次在麻省理工学院的MAC项目中得到一份暑期工作时，我为能够使用DEC PDP-10计算机而兴奋不已。毫无疑问，在这类计算机上使用汇编语言编程比在其他计算机上更有趣，因为它具有完成位测试、位屏蔽、字段操作和整数操作所需的丰富易用的指令集。尽管PDP-10计算机已停产多年，但它仍拥有不少狂热的推崇者，他们仍然运行着PDP-10的硬件和软件，他们使用个人计算机模拟它的指令集，运行它的操作系统和相关的应用程序，他们甚至编写新软件；现在，至少有一家网站是用PDP-10的模拟器来维护网页的（不要笑，这并不比使陈旧的汽车跑起来更愚蠢）。

就是在这时，在1972年的夏天，我有幸看到了麻省理工学院被称为HAKMEM[⊖]的最新研究备忘录。这是一本奇妙而不拘一格的技术小文集。这些课题的内容涉及到电路、数论等广泛领域，而其中最令我感兴趣的是那些具有独创性的编程小技巧一览表。这些小技巧通常描述一些漂亮而与众不同的对整数和位串的操作（例如计算一个字中值为1的位的数目），这些操作原本是使用长长的机器指令序列或循环来实现的，而这些小技巧展示了如何仅用两三个精心筛选的指令就能更聪明地完成同样工作；同时探索、揭示出这些指令间的相互关系。我犹如吃巨米花一样，而不是像吃糖果那样，如饥似渴地阅读这些编程小珍宝——这些小技巧充满着华美，充满着智慧，充满着诗一般的优雅。

我想，“一定是这样的，还有许多这样的小技巧。”多年来我的确收集了许多，也发现了一些。“应该把它们写成一本书。”

当我看到Hank Warren的手稿时真是欣喜若狂。他系统地收集了许多编程小技巧，对它们分门别类，并做了详尽的解释。虽然有一些小技巧是用机器指令的形式描述的，但这本书不仅仅适用于汇编语言程序员。本书论述计算机整数和位串的基本结构关系，给出对它们进行有效操作的高效技术。这些技术对C语言和Java语言同样有用。

许多关于算法和数据结构的书籍对于排序和搜索、维护散列表和二叉树、记录和指针的处理讲了许多非常复杂的技巧，但它们忽视了微小的数据块——位和位数组——所能完成的工作。仅仅使用二进制加法、减法和按位操作就可以完成的工作令人惊讶；进位链使一个位可以对它左边的所有位产生影响，这一事实使加法操作成为一个非常强大的数据处理操作，而对此我们没有给予足够的认识。

是的，应该有一本关于这些技巧的书。现在，它就在你的手中，它非常神奇。如果你要编写最优化编译器或者高性能的代码，就必须阅读这本书。也许不是每天都能用上这些技巧，但是如果发现自己某一天陷入了困境，似乎需要按一个字中的位做循环操作，或者需要对整

⊖ HAKMEM是“hacks memo”的缩写；一个PDP-10的36位字中可以存放6个6位字符，因此PDP-10高手们所用的大部分名字被限制在6个字符之内，我们已经习惯于从6个字符的名字缩写立即译解出原名。所以在当时，至少对编程高手来说，以“HAKMEM”命名该备忘录是有意义的。

数进行操作而觉得难以编码、或者真的需要让整数计算或位计算的内循环速度成倍提高，这时就需要查看这些小技巧。当然也可能纯粹是为了欣赏而读这本书。

Guy L. Steele, Jr.
马萨诸塞州，柏灵顿
2002年4月

前　言

用户注意：软件维护成本的增加与程序设计人员的创造力的平方成正比。

“程序员创造力第一法则”，Robert D. Bliss, 1992

本书是我多年来所收集的程序设计小技巧的集成，它们大部分只能应用于以2的补码的形式表示整数的计算机上。尽管当这些技巧与寄存器的长度相关时，我们假设机器是32位的，但是我们很容易将这些技巧运用到其他寄存器长度的计算机上。

本书不涉及大型的编程技巧，例如高级排序技术和编译器优化技术等；相反，它所讨论的是诸如计算一个字中值为1的位的数目这样的与计算机的字或指令相关的小技巧，这样的技巧通常是算术指令和逻辑指令的混合物。

本书自始至终都假设整数溢出中断被屏蔽，所以不会发生溢出中断。C语言、Fortran语言、Java语言运行在这样的环境下，但是Pascal语言和ADA语言的用户则要小心。

本书的描述是非形式化的。只有当算法不是显然的时才给出证明（有些也不证明）。我们使用计算机算术、“地板”函数、算术和逻辑操作的混合等来描述这些小技巧。在这些领域，证明通常相当困难并且难以表述。

为了减少印刷错误和疏忽，我们实际运行了很多算法。因此，尽管每一种计算机语言都有不尽人意的一面，但我们还是使用程序设计语言给出了这些算法。由于C语言的高知名度，我们把它作为高级语言使用，它既允许整数操作，也允许位串操作，并且我们有产生高质量目标代码的C编译器。

偶尔我们也使用机器语言。它使用三地址格式，主要是为了可读性。我们使用的汇编语言是RISC虚拟计算机的汇编语言。

我们尽量使用无分支代码。因为在许多机器上，分支会减慢指令提取，抑制指令的并行执行。分支的另一个问题是它们可能会抑制编译器的优化，例如指令调度、指令公用以及寄存器分配。也就是说，编译器可能会更有效地优化由若干大的基本块组成的程序，而对于由许多小基本块组成的程序进行优化的效果则可能不显著。

代码序列中也常出现小的立即值、与零的比较（而不是与其他数相比较）以及指令级的并行。尽管通过（从内存）查表可以使大部分代码更简洁，但本书不常提及查表法。这是因为相对于算术指令，其装入的代价越来越大，而且查表法通常不是很有趣（尽管它们通常很实用）。当然也有例外的情况。

最后，我要说明的是，书名中的“HACKER”一词指的是计算机狂热爱好者——是一些喜欢使用计算机做新鲜事或用更新更聪明的方法做一些旧事情的人。这些计算机狂热爱好者们通常技艺高超，但很可能并非专业的计算机程序员或程序设计者。他们的工作也许很有用，也许仅仅是一种游戏。不少坚定的计算机迷们写出这样的程序，在执行这个程序时它会生成

自己的一份拷贝[⊖]。这就是我们通常所说的电脑黑客。如果你是在找如何侵入他人电脑的技巧的话，在本书中是找不到的。

H. S. Warren, Jr.
约克镇，纽约
2002年2月

⊖ 据作者所知，用C语言写成的最短的这种程序是Vlad Taerov 和Rashit Fakhreyev所写的包含64个字符的程序：

```
main(a){printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);}",34);}
```

目　录

第1章 介 绍

1.1 记法

本书将普通算术的数学表达式与描述计算机操作的数学表达式区分开来。在“计算机算术”中，操作数是一些定长位串或定长位矢量。计算机算术中的表达式与普通算术表达式相似，但是，变量表示计算机寄存器的内容。计算机算术表达式的值仅仅是一个位串，没有特定含义。操作符用特定的方式来解释其操作数的含义。例如，比较操作符可以把它的操作数解释为带符号二进制整数，也可以解释成无符号二进制整数；这里的计算机算术记法使用不同的符号来明确比较的类型。

计算机算术与普通算术之间的一个主要区别是，在计算机算术中，我们使用模2^n来约简加法、减法和乘法的结果，其中，n是计算机的字长。另一个区别是计算机算术包括大量的操作，除了四个基本算术操作之外，计算机算术还包括逻辑与（and）、异或（exclusive or）、比较（compare）、左移位（shift left）等等。

除非特别声明，一般字长为32位，带符号整数以2的补码的形式表示。

计算机算术表达式与普通算术表达式的形式类似，只是表示计算机寄存器内容的变量用黑体字表记。这是矢量代数的习惯做法。我们把计算机的字看成由位组成的矢量。当常量代表计算机寄存器的内容时也用黑体字表记。（矢量代数中没有相应的表记方法，矢量代数表记常量的惟一方法是给出它的成分）。当一个常量代表一个指令的组成部分时，例如一个移位指令的立即字段，我们使用细体字来表记。

若一个如“+”这样的操作符带有黑体表记的操作数，那么该操作符就表示计算机的加法操作（“矢量加法”）。如果操作数是细体字，那么操作符就表示普通的标量算术运算。我们用细体的变量x去表示黑体变量$\mathbf{x}$在特定解释下的算术值，根据前后文决定把$\mathbf{x}$解释成带符号数还是无符号数。因此，如果$\mathbf{x}$=**0x80000000**、$\mathbf{y}$=**0x80000000**，那么在带符号整数的意义下$x=y=-2^{31}$，$x+y=-2^{32}$，而$x+y=0$。在这里，**0x80000000**是一个1位后面跟着31个0位组成的位串的十六进制表示。

对位的计数是从右侧开始的，最右侧（最小有效）位称作第0位。术语“位”、“半字节”、“字节”、“半字”、“字”和“双字”，分别表示长度1、4、8、16、32和64位。

我们使用赋值操作符（左箭头）及if语句这样的计算机代数来书写短小的代码片段。这里，计算机代数只不过是充当了与机器无关的汇编码的角色。

更长的或者更复杂的程序则用C++来书写。我们不使用C++的面向对象功能；程序本质上是带有C++注释风格的C程序。当区别不重要时，我们简单地用“C”来称呼这一语言。

对于C的完整说明超出了本书的范围，但我们在表1-1中列出了本书所用的C的大部分要素。这一表格主要是为那些对C语言以外的其他过程型编程语言有一定了解的读者准备的。表1-1还给出了计算机代数算术语言的操作符。操作符按着优先度从高到低排列。在优先度一栏中，

⊖ 此边栏的号码为该书原书页码，与书末索引中的页码相呼应。

L代表左结合，即，

$$a \cdot b \cdot c=(a \cdot b) \cdot c$$

而R代表右结合。这里，计算机代数中记号的优先度和结合性沿用C语言的约定。

除去表1-1所列出的记号外，我们也使用布尔代数和标准数学记号，并在必要的时候加以说明。

表1-1 C语言和计算机代数的表达式

优先度	C	计算机代数	说 明
	0x...	**0x**...,**0b**...	十六进制、二进制常数
16	a[k]		选择第k个分量
16		$x_0, x_1, \cdots$	不同的变量或位选择（根据文中说明）
16	f(x,...)	$f(x,\cdots)$	函数求值
16		abs(x)	绝对值（但是，abs(-2^{31})=-2^{31}）
16		nabs(x)	绝对值的负数
15	x++, x--		后置递增、后置递减
14	++x, --x		前置递增、前置递减
14	(类型名)x		类型转换
14R		x^k	x的k次幂
14	~x	$\neg x, \bar{x}$	按位否（1的补码）
14	!x		逻辑否（若x=**0**则为**1**，否则为**0**）
14	-x	$-x$	算术取负
13L	x*y	$x*y$	乘法，模字长
13L	x/y	$x \div y$	带符号整数除法
13L	x/y	$x \overset{u}{\div} y$	无符号整数除法
13L	x%y	rem(x,y)	$x \div y$的余数，可以为负，带符号参数
13L	x%y	remu(x,y)	$x \overset{u}{\div} y$的余数，无符号参数
		mod(x,y)	x模y，约简到区间[**0**,abs(y)−**1**]；带符号参数
12L	x+y, x-y	$x+y, x-y$	加法，减法
11L	x<<y, x>>y	$x<<y, x \overset{u}{>>} y$	带0填充的左移位和右移位（“逻辑”移位）
11L	x>>y	$x \overset{s}{>>} y$	带符号填充的右移位（“算术”或“代数”移位）
11L		$x \overset{rot}{<<} y, x \overset{rot}{>>} y$	循环左移位和右移位
10L	x<y,x<=y,x>y,x>=y	$x<y, x \leqslant y, x>y, x \geqslant y$	带符号比较
10L	x<y,x<=y,x>y,x>=y	$x \overset{u}{<} y, x \overset{u}{\leqslant} y, x \overset{u}{>} y, x \overset{u}{\geqslant} y$	无符号比较
9L	x == y, x!=y	$x=y, x \neq y$	相等，不等
8L	x & y	$x \,\&\, y$	按位与
7L	x^y	$x \oplus y$	按位异或
7L		$x \equiv y$	按位等值($\neg(x \oplus y)$)
6L	x | y	$x \mid y$	按位或
5L	x && y	$x \vec{\&} y$	条件与（若x=**0**则为**0**；否则，若y=**0**则为**0**，否则为**1**）
4L	x || y	$x \vec{\vee} y$	条件或（若$x \neq 0$则为**1**；否则，若$y \neq 0$则为**1**，否则为**0**）
3L		$x \,\|\|\, y$	连接
2R	x = y	$x \leftarrow y$	赋值

除“abs”、“rem”等函数外，计算机代数还使用其他函数。我们将在介绍这些函数时加以定义。

在C语言中，表达式x<y<z的意思是，先求x<y的值，然后将其0/1值结果与z比较。而在计算机代数中，表达式$x<y<z$意思是$(x<y)\&(y<z)$。

C语言有三种循环控制语句：while语句、do语句和for语句。while语句的书写格

式是:

while (*expression*) *statement*

对于这样的while语句，首先，求*expression*的值；如果*expression*的值为**true**(即非0)，执行*statement*，执行完毕后控制再次返回到对*expression*求值的操作。如果*expression*的值为**false**(即0)，则结束循环。

do语句的运行类似，只是在每次循环后进行条件检测。它的书写格式是:

do *statement* while (*expression*)

首先执行*statement*，然后对*expression*求值。如果值为**true**则重复这一过程，如果值为**false**则结束循环。

for语句的书写格式是:

for (e_1; e_2; e_3) *statement*

首先，运行e_1，它通常是赋值语句，然后对e_2求值，它通常是一个比较。如果e_2的值为**false**，则结束循环。如果e_2的值为**true**，则执行*statement*。最后，执行e_3，它通常是赋值语句，然后控制转移到对e_2再次求值。所以，常见的“do i=1 to n”书写成:

```
for (i=1; i<=n; i++)
```

(这是使用后置递增操作符的几种情况之一)。

1.2 指令集和运行时间模型

为了对算法做粗略的比较，我们使用与通用RISC计算机指令集类似的指令集对这些算法进行编码。这样的通用计算机包括Compaq Alpha、SGI MIPS以及IBM RS/6000等。我们使用三地址指令，并假定机器至少有16个通用寄存器。除非特别声明，寄存器的长度为32位。通用寄存器0的值总为0，而其他寄存器可以用于任意目的。

为简单起见，这里没有条件寄存器，也没有用于保存状态位（如“溢出”）等目的的“专用”寄存器。浮点操作也不在本书的讨论范畴。

有两种RISC:“基本RISC”拥有表1-2所列的指令；“完全RISC”除拥有“基本RISC”的所有指令外，还拥有表1-3所列的指令。

表1-2　基本RISC指令集

操作码助记符	操　作　数	说　　明
add, sub, mul, div, divu, rem, remu	RT, RA, RB	RT←RA op RB，其中，op分别是加、减、乘、带符号除、无符号除、带符号求余或无符号求余
addi, muli	RT, RA, I	RT←RA op I，其中，op分别是加或乘，I是一个16位带符号立即值
addis	RT, RA, I	RT←RA + (I || 0x0000)
and, or, xor	RT, RA, RB	RT←RA op RB，其中，op分别是按位与、按位或以及按位异或
andi, ori, xori	RT, RA, Iu	除最后操作数为16位无符号立即值外，其余同上条一样
beq, bne, blt, ble, bgt, bge	RT, target	分别当RT=0、RT≠0、RT<0、RT≤0、RT>0、RT≥0时分支到target（把RT解释为带符号整数）

（续）

操作码助记符	操 作 数	说 明
bt, bf	RT, target	当RT为true或false时分支到target；分别与bne和beq相同
cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu	RT, RA, RB	比较RA和RB：若结果为**false**则将0装入RT，若结果为**true**则将1装入RT。各助记符表示等于、不等于、小于等分支指令的比较。另外，后缀“u”表示相应比较是无符号比较
cmpieq, cmpine, cmpilt, cmpile, cmpigt, cmpige	RT, RA, I	除第二个比较操作数为16位带符号立即值外，其他同cmpeq等相应指令类似
cmpiequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu	RT, RA, Iu	除第二个比较操作数为16位无符号立即值外，其他同cmpltu等相应指令类似
ldbu, ldh, ldhu, ldw	RT, d(RA)	分别把从内存单元RA+d开始的一个无符号字节、带符号半字、无符号半字或一个字的内容装入到RT，其中，d为16位带符号立即值
mulhs, mulhu	RT, RA, RB	RT分别取RA和RB的带符号乘积和无符号乘积的高阶32位
not	RT, RA	RT←RA的按位1的补码
shl, shr, shrs	RT, RA, RB	RT←RA的左移位或右移位，移位量是RB的最右6位的值：shrs为符号填充，其余为0填充（位移量被视为模64）
shli, shri, shrsi	RT, RA, Iu	RT←RA的左移位或右移位，移位量由Iu的5位立即字段中给定的值决定
stb, sth, stw	RS, d(RA)	分别将RS的一个字节、半字或字存储于内存单元RA+d，其中，d是16位带符号立即值

在以上指令的简单说明中，作为源操作数的RA和RB指的是这些寄存器的内容。

实际的计算机拥有分支和调用子程序的链接，分支到一个寄存器中存放的地址（当子程序返回时及多分支执行完毕后），还可能拥有涉及专用寄存器的指令。当然，它会拥有若干特权指令，以及调用管理服务的指令，还可能拥有浮点指令。

表1-3中列出了RISC计算机可能拥有的其他计算指令。这些指令将在以后章节讨论。

表1-3 完全RISC的附加指令

操作码助记符	操 作 数	说 明
abs, nabs	RT, RA	RT分别取RA的绝对值和绝对值的负
andc, eqv, nand,nor, orc	RT, RA, RB	RT分别取RA和RB的补的按位与、RA和RB的按位等值、RA和RB的按位与非、RA和RB的按位或非，以及RA和RB的补的按位或
extr	RT, RA, I, L	从RA中提取第I位到第I+L-1位的位串，然后右对齐放入RT中，用0填充剩余位
extrs	RT, RA, I, L	与extr类似，不同的是用符号填充剩余位
ins	RT, RA, I, L	将RA的第0位到第L-1位的位串插入到RT的第I位到第I+L-1位
nlz	RT, RA	RT取RA中前导0的数目，取值范围为0到32
pop	RT, RA	RT取RA中值为1的位的数目，取值范围为0到32

（续）

操作码助记符	操 作 数	说　明
ldb	RT, d(RA)	把从内存单元RA+d开始的一个带符号字节的内容装入到RT，其中，d为16位带符号立即值
moveq, movne, movlt, movle, movgt, movge	RT, RA, RB	分别当RA=0、RA≠0、RA<0、RA≤0、RA>0、RA≥0时进行RT←RB，否则RT不变
shlr, shrr	RT, RA, RB	RT分别取RA的循环左移位和循环右移位的结果，移位量由RB的最右5位的值决定
shlri,shrri	RT,RA,Iu	RT分别取RA的循环左移位和循环右移位的结果，移位量由Iu的5位立即字段中给定的值决定
trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu	RA, RB	分别当RA=RB、RA≠RB、RA<RB、RA≤RB、RA>RB、RA≥RB、$RA \overset{u}{<} RB$、$RA \overset{u}{\leq} RB$、$RA \overset{u}{>} RB$、$RA \overset{u}{\geq} RB$时执行中断
trpieq, trpine, trpilt, trpile, trpigt, trpige	RA, I	除第二个比较操作数为16位带符号立即值外，其他同trpeq等相应指令类似
trpiequ, trpineu, trpiltu, trpileu, trpigtu, trpigeu	RA, Iu	除第二个比较操作数为16位无符号立即值外，其他同trpltu等相应指令类似

可以很方便地为机器的汇编程序提供几个“扩展助记符”。这些助记符类似于宏，它们通常可以展开成单个指令。表1-4列出了一些这样的扩展助记符。

表1-4　扩展助记符

扩展助记符	展　开	说　明
b　target	beq R0,target	无条件分支
li　RT,I	见下面正文	立即装入，$-2^{31} \leq I < 2^{32}$。
mov　RT,RA	ori　RT, RA, 0	将寄存器RA传送到RT
neg　RT,RA	sub　RT, R0, RA	按2的补码取负
subi RT,RA,I	addi RT, RA, -I	立即减（$I \neq -2^{15}$）

立即装入指令根据立即值*I*的需要可能展开成一个指令，也可能展开成两个指令。例如，如果$0 \leq I < 2^{16}$，则可以使用R0和*I*的立即或（ori）。如果$-2^{15} \leq I < 0$，则可以使用R0和*I*的立即加（addi）。如果*I*的最右16位均为0，则可以使用移位立即加（addis）。在其他情况下则需要两个指令，例如，在ori指令后跟着一个addis指令。（或者，最后一种情况也可以使用内存到寄存器的装入操作，然而，为了评估执行时间开销和空间开销，我们假设使用两个初等算术指令。）

当然，不好判断哪些指令应该属于基本RISC指令集，哪些应该属于完全RISC指令集。无符号除和求余指令也许应该属于完全RISC指令集。带符号右移位指令也是一个值得商榷的指令，因为在SPEC基准中它的使用频率很低。麻烦的是，在C语言中，偶然会用到这些指令，例如，既可能要对无符号操作数做除法操作，也可能要对带符号操作数做除法操作，既可能对带符号量（int）又可能对无符号量做右移位操作。顺便提一下，不能使用带符号右移位

（通常称为算术右移位）来实现带符号整数除以2的幂的运算；如果被除数是负数，而且有非零位被移出，那么需要在移位结果上加1。

属于基本RISC指令集还是完全RISC指令集的判断还包括许多其他诸如此类的难点，但是我们对此不一一说明。

我们把指令局限于带两个源寄存器和一个目标寄存器来简化计算机（例如，寄存器文卷（register file）最多需要两个输入接口（read port）和一个输出接口（write port））。由于编译器不需要处理多目标指令，它也可以简化优化编译器的实现。这样处理的代价是，需要同时求两个数的商和余数的程序必须执行两个指令（除指令和求余指令），而这样的需求并非罕见。普通计算机的除法把余数作为一种副产品，所以很多计算机通过运行一次除法可以得到这两个结果。同样的讨论也适用于两个字的双字积。

moveq这样的条件传送指令表面上只有两个源操作数，但是在某种意义上，它们有三个源操作数。因为指令的结果依赖于RT、RA和RB的值，一个无序执行指令的计算机必须在这些指令中同时把RT作为“使用”和“设置”的对象来处理。也就是说，需要在设置RT的条件传送指令之前执行设置RT的指令，必须按着这种顺序执行，第一个指令的结果不能忽略不计。因此，计算机的设计者可能选择省略条件传送指令，从而避免去考虑（逻辑上）需要三个源操作数的指令。而另一方面，条件传送指令可以节省分支。

指令的格式与本书的目的无关，但是上述RISC指令集以及浮点指令和若干管理指令可以
8 在一台拥有32个32位通用寄存器（5位寄存器字段）的计算机上实现。通过把比较、装入、存储和中断指令的立即字段减少到14位，这些指令可以在拥有64个通用寄存器（6位寄存器段）的计算机上实现。

执行时间

我们假设除了乘法、除法和求余指令之外，执行每个指令的开销是一个周期，而对乘、除和求余指令不假设任何特定的执行时间。无论是否发生分支，分支指令都要使用一个周期。

立即装入指令需要使用一到两个周期，这依赖于在寄存器中产生常量需要一个还是两个初等算术指令。

尽管本书不常用装入和存储指令，我们假设它们需要一个周期，并且忽视任何装入延迟（就是在运算单元中从一个装入指令的完成到下一个指令获得所需数据的时间间隔）。

然而，仅知道所有算术和逻辑指令所使用的周期数对估计一个程序的执行时间是不够的。装入延迟和在提取指令上的延迟都足以使程序的执行减慢。尽管这些延迟非常重要，而且其重要性还在日益增加，但本书不讨论这些因素。改进执行时间的另外一个因素就是所谓的指令级并行，许多当代RISC芯片，特别是“高端”计算机的芯片，都可以实现指令级并行。

这些计算机都有多个执行单元以及足够的指令分配能力，可以并列执行相互独立的指令（即，当每个指令都不使用其他指令的结果，也不设置同一个寄存器或状态位时）。因为今天这种能力相当普遍，书中会经常指明相互独立的操作。因此，我们可以说某某公式可以这样编码：它需要8个指令，而在一台带有无限制指令级并行的计算机上需要5个周期。这意味着如果用适当的次序安排指令的执行（进行调度），一台带有足够数目的加法器、移位器、逻辑单元和寄存器的计算机理论上讲可以用5个周期来执行这个代码。

我们对此不再过多论述，因为计算机的指令级并行能力存在着很大的差异。例如，1992年产的IBM RS/6000处理器有一个三输入端加法器，能够并行执行两个连续的加类型指令，

甚至当一个指令的结果要供给另一个指令使用时也可以并行执行（例如，供给比较使用的加，或供给装入指令基寄存器值使用的加）。作为反例，考虑一台简单的计算机，例如廉价的嵌入式计算机，它的寄存器文卷只有一个输入端口。通常，对于有两个寄存器输入操作数的指令，这样的机器需要一个额外的周期再次读取寄存器文卷。然而，假设它有一个旁路，使得如果一个指令为接下来的指令提供操作数，那么那个操作数无需读取寄存器文卷即可使用。在这样的机器上，不使用并行代码，而让每个指令都为下一个指令提供数据，这种方式更有优势。 9

第2章　基　　础

2.1　操作最右侧位

在后面几章中我们会看到对本节中某些公式的应用。

下面的公式将一个字的最右侧的1位改成0位，如果没有1位则生成所有的位都为0的字（例如，0101 1000=>0101 0000）：

$$x \,\&\, (x-1)$$

这个公式可以用来检查一个无符号整数是否为2的幂；在使用这个公式之后，对运算结果做一个0-检测。

类似地，下面的公式可以用来检测一个无符号整数是否为2^n-1的形式（包括0或所有位均为1的字的情况）：

$$x \,\&\, (x+1)$$

使用下面的公式析出（isolate）最右侧的1位，如果没有1位则生成所有位均为0的字（例如，0101 1000=>0000 1000）：

$$x \,\&\, (-x)$$

使用下面的公式析出最右侧的0位，如果没有0位则生成所有位均为0的字（例如，1010 0111 => 0000 1000）：

$$\neg x \,\&\, (x+1)$$

使用下面的公式之一构造识别后缀0的掩码，如果$x=0$则生成所有位都为1的字（例如，0101 1000=>0000 0111）：

$$\neg x \,\&\, (x-1), \text{ 或}$$

$$\neg (x \mid -x), \text{ 或}$$

$$(x \,\&\, -x) - 1$$

其中，第一个公式具有指令级并行性质。

使用下面的公式构造识别最右侧的1位和后缀0的掩码，如果$x=0$则生成所有的位都为1的字（例如，0101 1000=>0000 1111）：

$$x \oplus (x-1)$$

使用下面的公式来向右传播最右侧的1位，如果$x=0$则生成所有的位都为1的字（例如，0101 1000=>0101 1111）：

$$x \mid (x-1)$$

使用下面的公式将最右侧连续的1位串改成0位串（例如，0101 1000=>0100 0000）：

$$((x \mid (x-1)) + 1) \,\&\, x$$

对于$j>k>0$，这一公式可以用来检查一个非负整数是否具有2^j-2^k的形式；使用这个公式之后，对运算结果做0-检测。

在以下意义下，上面这些公式都具有对偶性。交换描述中的1和0，公式会怎样呢。那么，在公式中，用$x+1$替换$x-1$，用$x-1$替换$x+1$，用$\neg(x+1)$替换$-x$，用&替换|，用|替换&；而x和$\neg x$不变。那么，就将得出一个新的正确的描述和公式。例如，本节第一个公式的对偶可如下解读：

下面的公式将一个字的最右侧的0位改成1位，如果没有0位则生成所有位都为1的字：（例如，1010 0111⇒1010 1111）：

$$x \mid (x+1)$$

对于给定函数是否能用一连串的加、减、与、或、非指令来实现，有一种简单检测方式[War]。当然，我们也可以使用其他能用基本指令集组成的指令来扩展指令集，例如，可以使用固定数量的左移位指令（它等价于一系列的加），也可以使用乘指令。然而，我们把不能用上述指令集组成的指令排除在外。下面的定理给出检测的方法：

定理 将字映射到字的函数可以用字并行加、减、与、或、非指令实现，当且仅当函数结果的每一位只依赖于每个输入操作数的相应位以及相应位右侧的位。

也就是说，给定一个函数，尝试通过查看每个输入操作数的最右侧的位来计算结果的最右侧的位。然后，通过查看每个输入操作数的最右侧的两个位来计算结果的右数第二位，以此类推。如果成功了，那么就可以用一连串加、与等指令来实现这个函数。如果这个函数不能按上述方式计算，那么这个函数就不能用一连串这样的指令实现。

定理后半部分的陈述非常有趣，它是下述事实的简单倒置，即，加、减、与、或、非这
12 些函数都可以按从右到左的方式计算，所以它们的任意组合也一定具有同样的性质。

我们需要一个结构来理解上述定理的“当”部分，这个结构解释起来有点麻烦，我们使用特例来加以说明。假设一个二元函数（带有变量x和y）具有从右到左可计算的性质，并假设结果r的第2位由下面的公式给出：

$$r_2 = x_2 \mid (x_0 \,\&\, y_1) \tag{2-1}$$

我们从右到左（从0到31）对位进行计数。因为结果的第2位是输入操作数的第2位及其右侧各位的函数，所以结果的第2位是“从右到左可计算的”。

如下所示，排列计算机的字x，对x左移位2个位，对y左位移1个位，再加一个析出第2位的掩码。

$$\begin{array}{cccccccc}
x_{31} & x_{30} & \cdots & x_3 & x_2 & x_1 & x_0 \\
x_{29} & x_{28} & \cdots & x_1 & x_0 & 0 & 0 \\
y_{30} & y_{29} & \cdots & y_2 & y_1 & y_0 & 0 \\
0 & 0 & \cdots & 0 & 1 & 0 & 0 \\
0 & 0 & \cdots & 0 & r_2 & 0 & 0
\end{array}$$

现在，对第二行和第三行做字并行与，并将这一结果与第一行（按等式(2-1)）做或，再将刚才所得结果与掩码（第四行）做与。最终得到一个第2位为所求结果而其他位都是零的字。对结果的其余位进行类似的计算，对得到的32个字做或，其结果就是所求的函数。

这一结构不能产生高效的程序，它只表明使用基本指令集中的指令可以实现计算这一函数的程序。

使用这一定理，我们立即就可以看出，不存在将一个字的最左侧的1位改变成0的基本指令序列，因为当我们要看某个特定的1位是否该被改变为0时，我们必须看它的左侧来确定它是否是最左侧的1位。类似地，也不存在执行右移位，或循环移位，或移位量不定的左移位；不存在计算一个字的后缀0的数目的指令序列（要计算后缀0的数目，如果后缀0的数目是奇数，那么结果的最右侧位将是1，而我们必须看最右位置的左侧才能决定后缀0数目的奇偶性）。

上面所讨论的这种位的移动操作的一个新应用是，寻找与一个给定的数目有相同数目的1位的下一个大数的问题。读者可能会迷惑，“为什么有人会想要计算它呢？”它在使用位串来表示子集时有用。把一个集合的所有可能元素列成一个线性序列，使用字或字的序列来表示子集，当集合的成员i属于该子集时设置字或字序列的第i位为1。这样，集合的并可以用位串的逻辑或来计算，集合的交可以用位串的逻辑与来计算，等等。 13

也许想对给定大小的所有子集进行迭代。如果有一个把给定的子集映射到下一个具有相同数目的1位的更大的数（把这个子集串看成整数）的函数的话，很容易实现这种迭代。

进行这种操作的简明算法是由R. W. Gosper设计的[HAK，item 175]⊖。给定一个代表子集的字x，寻找x中最右侧连续的1位组及其后面的0，“递增”这个量到下一个具有相同数目的1位的值。例如，串xxx0 1111 0000变成xxx1 0000 0111，其中xxx表示任意的位。这个算法首先使用$s=x\&-x$识别出x中的“最小”1位，得到0000 0001 0000。把这个值加到x上，得到r=xxx1 0000 0000。这里的1位就是结果中的一个位。对于其他的位，我们需要生成一个$n-1$个右对齐的1位串，其中n是x中最右侧1位组的大小。首先形成r和x的一个异或，这个异或在我们的例子中给出0001 1111 0000。

这里多出两个1位，并且需要做右对齐。这可以通过用s除它来实现右对齐（s是2的幂），然后向右移位2个位来丢掉两个不要的1位。最终结果就是这个值和r的或。

使用计算机代数的记法，结果是下面操作中的y

$$\begin{aligned} s &\leftarrow x \,\&\, -x \\ r &\leftarrow s + x \\ y &\leftarrow r \mid (((x \oplus r) \overset{u}{\gg} 2) \overset{u}{\div} s) \end{aligned} \tag{2-2}$$

图2-1给出了完整的C语言过程。它的执行需要7个基本RISC指令，其中有一个指令是除运算（使用这一过程时不要用x=0，否则将导致除0错误。）

```
unsigned snoob(unsigned x) {
   unsigned smallest, ripple, ones;
                                   // x = xxx0 1111 0000
   smallest = x & -x;              //     0000 0001 0000
   ripple = x + smallest;          //     xxx1 0000 0000
   ones = x ^ ripple;              //     0001 1111 0000
   ones = (ones >> 2)/smallest;    //     0000 0000 0111
   return ripple | ones;           //     xxx1 0000 0111
}
```

图2-1　下一个具有同样数目的1位的更大的数 14

⊖ 这个算法的变体参见[H&S] 7.6.7节。

如果除运算比较慢，而你有计算后缀0数目的函数ntz(x)、计算前导0数目的函数nlz(x)或种群计数函数pop(x)（pop(x)是x中1位的数目）的快捷方法的话，那么，可以用下列任何一个操作替换公式(2-1)的最后一行。

$$y \leftarrow r \mid ((x \oplus r) \ggg (2 + \mathrm{ntz}(x)))$$

$$y \leftarrow r \mid ((x \oplus r) \ggg (33 - \mathrm{nlz}(s)))$$

$$y \leftarrow r \mid ((\mathbf{1} \ll (\mathrm{pop}(x \oplus r) - 2)) - 1)$$

2.2 结合逻辑操作的加运算

我们假设读者熟悉普通代数和布尔代数的基本恒等式。下面是一些结合了逻辑操作的加运算和减运算的恒等式：

$$\begin{aligned}
\text{a.} \quad -x &= \neg x + 1 \\
\text{b.} \quad &= \neg(x - 1) \\
\text{c.} \quad \neg x &= -x - 1 \\
\text{d.} \quad -\neg x &= x + 1 \\
\text{e.} \quad \neg -x &= x - 1 \\
\text{f.} \quad x + y &= x - \neg y - 1 \\
\text{g.} \quad &= (x \oplus y) + 2(x \,\&\, y) \\
\text{h.} \quad &= (x \mid y) + (x \,\&\, y) \\
\text{i.} \quad &= 2(x \mid y) - (x \oplus y) \\
\text{j.} \quad x - y &= x + \neg y + 1 \\
\text{k.} \quad &= (x \oplus y) - 2(\neg x \,\&\, y) \\
\text{l.} \quad &= (x \,\&\, \neg y) - (\neg x \,\&\, y) \\
\text{m.} \quad &= 2(x \,\&\, \neg y) - (x \oplus y) \\
\text{n.} \quad x \oplus y &= (x \mid y) - (x \,\&\, y) \\
\text{o.} \quad x \,\&\, \neg y &= (x \mid y) - y \\
\text{p.} \quad &= x - (x \,\&\, y) \\
\text{q.} \quad \neg(x - y) &= y - x - 1 \\
\text{r.} \quad &= \neg x + y \\
\text{s.} \quad x \equiv y &= (x \,\&\, y) - (x \mid y) - 1 \\
\text{t.} \quad &= (x \,\&\, y) + \neg(x \mid y) \\
\text{u.} \quad x \mid y &= (x \,\&\, \neg y) + y \\
\text{v.} \quad x \,\&\, y &= (\neg x \mid y) - \neg x
\end{aligned}$$

15

可以重复运用等式(d)来得到$-\neg-\neg x = x+2$，等等。类似地，从等式(e)我们有$\neg-\neg- x = x-2$。因此我们可以使用这两种求补形式加上或减去任何常数。

等式(f)是等式(j)的对偶形式，在这里等式(j)表明了众所周知的从加法器构建减法器的关系。

等式(g)和等式(h)来自于HAKMEM备忘录[HAK，item23]。等式(g)通过首先忽略进位进行求和（$x \oplus y$），然后再加上进位来求和。等式(h)简单地修改加的操作数，使得在任何位的位

置上都不会发生0+1的组合：它被1+0取代。

很显然，在通常的二进制数的加法中，每个位为1和0的可能性相同并相互独立，所以在每个位置上发生进位的概率约为0.5。然而，如果利用等式(g)预先处理输入来构建加法器，那么进位的概率大约是0.25。这一观察对构建加法器也许没有价值，因为构建加法器的重要特征是进位所必须经过的逻辑回路的最大数目，而使用等式(g)只能减小一个进位传播的回路阶段数目。

等式(k)和等式(l)是等式(g)和等式(h)的减法对偶。也就是说，等式(k)可以解释成首先忽略借位求差（$x \oplus y$），然后再减去借位。类似地，等式(l)简单地修改减法操作数使得在任何位上都不会发生1−1的组合：它被0−0取代。

等式(n)表明如何仅用三个基本RISC指令实现异或。仅用与、或、非逻辑需要4个指令（$(x|y)\&\neg(x\&y)$）。类似地，等式(u)和等式(v)表明如何用其他三个基本指令实现与和或，而使用DeMorgan律则需要4个指令。

2.3　逻辑和算术表达式中的不等式

当二元逻辑表达式的值可以解释成无符号整数时，很容易得到二元逻辑表达式中的不等式。这里给出两个例子：

$$(x \oplus y) \overset{u}{\leqslant} (x \mid y), \text{ 和}$$

$$(x \,\&\, y) \overset{u}{\leqslant} (x \equiv y)$$

这些不等式可以由表2-1列出的所有二元逻辑操作列表得到。

设$f(x,y)$和$g(x,y)$代表表2-1的两列。如果对于$f(x,y)$为1的每一行，$g(x,y)$也为1，那么对所有的(x,y)，有$f(x,y) \overset{u}{\leqslant} g(x,y)$。显然，这可扩展到字并行逻辑操作。我们很容易从表中读取诸如$(x\&y) \overset{u}{\leqslant} x \overset{u}{\leqslant} (x|\neg y)$这样的关系（它们大部分是显然的）。此外，如果两列中有一行一个项是0而另一项是1，而另一行的项分别是0和1，那么相应的逻辑表达式之间没有不等关系。所以，对于所有的二元逻辑函数f和g，$f(x,y) \overset{u}{\leqslant} g(x,y)$是否成立的问题完全可以很容易地解决。 16

表2-1　16个二进制逻辑操作

x	y	0	$x \,\&\, y$	$x \,\&\, \neg y$	x	$\neg x \,\&\, y$	y	$x \oplus y$	$x \mid y$	$\neg(x \mid y)$	$x \equiv y$	$\neg y$	$x \mid \neg y$	$\neg x$	$\neg x \mid y$	$\neg(x \,\&\, y)$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

在处理这些关系时要小心。例如，对于普通的算术，如果$x+y<a$且$z<x$，则$z+y<a$。但是如果将“+”用或替换，那么这一结论就不再成立。

逻辑和算术混合表达式中的不等式更有趣。下面是其中的一小部分：

a.　　$(x \mid y) \overset{u}{\geqslant} \max(x,y)$

b. $(x \,\&\, y) \overset{u}{\leqslant} \min(x, y)$

c. $(x \mid y) \overset{u}{\leqslant} x + y$ 如果和没有溢出

d. $(x \mid y) \overset{u}{>} x + y$ 如果和溢出

e. $|x - y| \overset{u}{\leqslant} (x \oplus y)$

上述不等式中除不等式(e)之外都很容易证明。我们用$|x-y|$表示$x-y$的绝对值，在无符号数的范畴，可以用$\max(x,y)-\min(x,y)$计算绝对值$|x-y|$。可以通过对x和y的长度应用归纳法来证明不等式(c)（展开左边要比展开右边容易证明）。

2.4 绝对值函数

如果计算机中没有计算绝对值的指令，那么绝对值的计算可以用3或4个无分支指令来完成。首先，计算$y \leftarrow x \overset{s}{>>} 31$，然后使用下面的公式之一：

abs	nabs
$(x \oplus y) - y$	$y - (x \oplus y)$
$(x + y) \oplus y$	$(y - x) \oplus y$
$x - (2x \,\&\, y)$	$(2x \,\&\, y) - x$

17 其中“$2x$”表示$x+x$或$x<<1$。

如果有与取值为±1的变量相乘的快速乘法，那么下面的式子将计算绝对值：

$$((x \overset{s}{\gg} 30) \mid 1) * x$$

2.5 符号扩展

“符号扩展”的意思就是认定一个字中的某个位是符号位，我们希望把这个符号位向左传播，忽略其他位。做这一工作的标准方法就是先做*逻辑左移位*，然后再做*带符号右移位*。然而，如果计算机上没有这些指令或它们比较慢的话，那么我们可以用下面所给指令中的一个来实现这一操作，下面说明如何把第7位向左传播：

$$((x + \mathbf{0x00000080}) \,\&\, \mathbf{0x000000FF}) - \mathbf{0x00000080}$$

$$((x \,\&\, \mathbf{0x000000FF}) \oplus \mathbf{0x00000080}) - \mathbf{0x00000080}$$

上式中的“+”也可以是“−”或“⊕”。如果知道不需要的高阶位都是0的话，那么上面的第二个公式非常有用，因为这样的话就可以省略与。

2.6 用无符号右移位实现带符号右移位

如果计算机中没有*带符号右移位*指令，那么可以用下面所给的公式计算带符号右移位。第一个公式来自[GM]，第二个公式也基于同样的思路。假设计算机有模64移位，前四个公式对$0 \leqslant n \leqslant 31$成立，最后一个公式对$0 \leqslant n \leqslant 63$成立。在“对移位量做与逻辑移位同样的模操作”的意义下，最后一个公式对任何n都“成立”。

当n为变量时，下面的每个公式都需要5到6个基本RISC指令。

$$((x + \mathbf{0x80000000}) \overset{u}{\gg} n) - (\mathbf{0x80000000} \overset{u}{\gg} n)$$

$$t \leftarrow \mathbf{0x80000000} \overset{u}{\gg} n; \qquad ((x \overset{u}{\gg} n) \oplus t) - t$$

$$t \leftarrow (x \,\&\, \mathbf{0x80000000}) \overset{u}{\gg} n; \; (x \overset{u}{\gg} n) - (t + t)$$

$$(x \overset{u}{\gg} n) \mid (-(x \overset{u}{\gg} 31) \ll \mathbf{31} - n)$$

$$t \leftarrow -(x \overset{u}{\gg} 31); \qquad ((x \oplus t) \overset{u}{\gg} n) \oplus t$$

在前两个公式中，可以用**1<<31−*n***替换**0x80000000** $\overset{u}{\gg}$ *n*。 18

如果*n*是常数，那么在许多计算机上前两个公式只需要3个指令。如果*n*=31，那么利用 $-(x \overset{u}{\gg} 31)$，只需用2个指令就可以实现带符号右移位。

2.7 符号函数

符号函数，也称正负号函数，定义如下:

$$\operatorname{sign}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

在大多数计算机上，可以用4个指令来计算这一函数[Hop]:

$$(x \overset{s}{\gg} 31) \mid (-x \overset{u}{\gg} 31)$$

如果没有带符号右移位指令，那么可以使用2.6节最后给出的式子替代，得到下面的（5指令）对称公式:

$$-(x \overset{u}{\gg} 31) \mid (-x \overset{u}{\gg} 31)$$

使用比较谓词指令可以得到如下3指令解:

$$\begin{aligned} &(x > 0) - (x < 0), \quad \text{或} \\ &(x \geqslant 0) - (x \leqslant 0). \end{aligned} \tag{2-3}$$

最后，注意，公式$(-x \overset{u}{\gg} 31)-(x \overset{u}{\gg} 31)$几乎能够正常工作；它只对$x=-2^{31}$失败。

2.8 三值比较函数

三值比较函数是符号函数的一般化，定义如下:

$$\operatorname{cmp}(x, y) = \begin{cases} -1, & x < y, \\ 0, & x = y, \\ 1, & x > y. \end{cases}$$

它有带符号和无符号两种形式，除非特别声明，本节的内容对两者均适用。 19

下式显然是公式(2-3)的一般化，它给出比较谓词指令的3指令解:

$$\begin{aligned} &(x > y) - (x < y), \quad \text{或} \\ &(x \geqslant y) - (x \leqslant y) \end{aligned}$$

下面给出PowerPC机对于无符号整数的解 [CWG]。在这类计算机上，“进位”是“非借位”。

```
subf   R5,Ry,Rx    # R5 <-- Rx - Ry.
subfc  R6,Rx,Ry    # R6 <-- Ry - Rx, set carry.
subfe  R7,Ry,Rx    # R7 <-- Rx - Ry + carry, set carry.
subfe  R8,R7,R5    # R8 <-- R5 - R7 + carry, (set carry).
```

如果局限于基本RISC指令，那么似乎没有计算这个函数的更好方法。比较谓词$x<y$，$x\leqslant y$，等等都需要5个指令（参见2.11节），导致一个需要大约12个指令的解（在计算$x<y$和$x>y$时使用一些通用的计算）。在基本RISC指令中，使用比较和分支指令也许更好些（如果比较指令可以公用的话，最坏情况需要6个指令）。

2.9 符号传递

在Fortran中ISIGN被称为符号传递函数，它的定义如下：

$$\mathrm{ISIGN}(x, y) = \begin{cases} \mathrm{abs}(x), & y \geqslant 0, \\ -\mathrm{abs}(x), & y < 0. \end{cases}$$

在大多数计算机上，可以使用四个指令来计算这一函数（模2^{32}）：

$t \leftarrow y \gg 31;$

$$\begin{aligned} \mathrm{ISIGN}(x, y) &= (\mathrm{abs}(x) \oplus t) - t \\ &= (\mathrm{abs}(x) + t) \oplus t \end{aligned}$$

$t \leftarrow (x \oplus y) \gg 31;$

$$\begin{aligned} \mathrm{ISIGN}(x, y) &= (x \oplus t) - t \\ &= (x + t) \oplus t \end{aligned}$$

2.10 对“0意味着2^n”字段的解码

有时0或负数对一个量来说没有意义，所以在n位字段中，可以用0表示2^n，而对于非零数则用它自身的规范二元解释来编码。PowerPC的立即装入字符串字指令（lswi）的长度字段就是一个这样的例子，这个长度字段占据5个位。当长度是一个立即量时，装入零字节的指令没有用处，但是能够装入32字节却很有用。我们可以对长度字段进行编码，用值0到31来表示长度从1到32，但当处理器还必须支持一个拥有可变长度的相应指令时（例如，PowerPC的lswx指令），“0意味着32”将导致更简单的逻辑，我们可以直接实现其二进制编码。

20

很容易把范围在1到2^n的整数编码到“零意味着2^n”编码：只需用2^n-1屏蔽这个整数即可。不使用测试－分支时进行解码则不那么简单，但是，下面是对于3位字段的几个可行的方法。它们都需要3个指令，不包括可能需要的常量装入。

$((x-1)\ \&\ 7)+1$	$((x+7)\mid -8)+9$	$8-(-x\ \&\ 7)$
$((x+7)\ \&\ 7)+1$	$((x+7)\mid 8)-7$	$-(-x\mid -8)$
$((x-1)\mid -8)+9$	$((x-1)\ \&\ 8)+x$	

2.11 比较谓词

“比较谓词”是比较两个量的函数，如果比较为**真**，则生成1，如果比较为**假**，则生成0。下面我们给出将比较的结果放入符号位的无分支表达式。为了生成某种语言（例如，C语言）所使用的1/0值结果，后面需要跟一个右移位31个位的代码。为了生成某些其他语言（例如，

Basic语言）所用的-1/0值结果，后面则需跟一个带符号右移位31个位的代码。

当然，这些公式对诸如MIPS、Compaq Alpha以及我们的RISC计算机模型等没有用处，因为这些计算机具有这样的比较指令，它们可以直接计算许多谓词并把其0/1值结果放入某个通用寄存器。

计算绝对值的负值的计算机指令是唾手可得的。我们把这个函数称为“nabs”。与绝对值不同，由于它不会产生溢出，因此它是良定义的。没有“nabs”但有更常用的“abs”的计算机可以把-abs(*x*)当作nabs(*x*)使用。如果*x*是最大负数，这将产生两次溢出，但是结果是正确的。（假定最大负数的绝对值和它的负值是它本身。）因为某些计算机既没有“abs”也没有“nabs”，所以还给出了一个不使用这两个函数的方法。

函数“nlz”的值是它的自变量中前导零的数目。“doz”（差或零）函数的说明参见2.18节。 21

$x = y$:　$\mathrm{abs}(x - y) - \mathbf{1}$

$\mathrm{abs}(x - y + \mathbf{0x80000000})$

$\mathrm{nlz}(x - y) \ll 26$

$-(\mathrm{nlz}(x - y) \overset{u}{\gg} 5)$

$\neg(x - y \mid y - x)$

$x \neq y$:　$\mathrm{nabs}(x - y)$

$\mathrm{nlz}(x - y) - \mathbf{32}$

$x - y \mid y - x$

$x < y$:　$(x - y) \oplus [(x \oplus y) \,\&\, ((x - y) \oplus x)]$

$(x \,\&\, \neg y) \mid ((x \equiv y) \,\&\, (x - y))$

$\mathrm{nabs}(\mathrm{doz}(y, x))$　[GSO]

$x \leqslant y$:　$(x \mid \neg y) \,\&\, ((x \oplus y) \mid \neg(y - x))$

$((x \equiv y) \overset{s}{\gg} 1) + (x \,\&\, \neg y)$　[GSO]

$x \overset{u}{<} y$:　$(\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y))$

$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y))$

$x \overset{u}{\leqslant} y$:　$(\neg x \mid y) \,\&\, ((x \oplus y) \mid \neg(y - x))$

对于$x>y$、$x \geqslant y$等的比较，交换上述$x<y$、$x \leqslant y$等公式中的x和y即可。与**0x8000 0000**的和可以用任何能够颠倒（x、y或$x-y$中）高阶位的指令取代。

谓词$x<y$可由$x/2-y/2$的符号给出，而且表达式中的减法不产生溢出。通过这一观察可以得到另外一组公式。对于移位丢掉了重要信息的情况，我们可以通过减1来修正结果。公式如下所示：

$x < y$:　$(x \overset{s}{\gg} 1) - (y \overset{s}{\gg} 1) - (\neg x \,\&\, y \,\&\, \mathbf{1})$

$x \overset{u}{<} y$:　$(x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - (\neg x \,\&\, y \,\&\, \mathbf{1})$

在大多数计算机上，这些公式的执行需要7个指令（如果有与非则需要6个指令），这并不比上面所给的公式好（上面的公式根据逻辑指令集中指令的多寡需要5到7个指令）。

上述包括“nlz”的公式是[Shep]给出的，他的谓词$x=y$的公式特别有用，因为它的一个很

小的变化形式可以仅用3个指令计算该谓词的1/0值结果:

22

$$\mathrm{nlz}(x - y) \overset{u}{\gg} 5$$

与0的带符号比较非常常用，值得特别提出。下面就是一些这样的公式，这些公式大部分可以从上面直接得到。在这里，结果还是放入符号位。

$x = 0$:	$\mathrm{abs}(x) - 1$	
	$\mathrm{abs}(x + \mathbf{0x80000000})$	
	$\mathrm{nlz}(x) \ll 26$	
	$-(\mathrm{nlz}(x) \overset{u}{\gg} 5)$	
	$\neg(x \mid -x)$	
	$\neg x \,\&\, (x - 1)$	
$x \neq 0$:	$\mathrm{nabs}(x)$	
	$\mathrm{nlz}(x) - 32$	
	$x \mid -x$	
	$(x \overset{u}{\gg} 1) - x$	[CWG]
$x < 0$:	x	
$x \leqslant 0$:	$x \mid (x - 1)$	
	$x \mid \neg -x$	
$x > 0$:	$x \oplus \mathrm{nabs}(x)$	
	$(x \overset{s}{\gg} 1) - x$	
	$-x \,\&\, \neg x$	
$x \geqslant 0$:	$\neg x$	

通过让带符号操作数向上偏移2^{31}并将结果作为无符号整数处理，就可以从相应的无符号比较得到带符号比较。逆变换同样正确。因此我们有:

$$x < y \;=\; x + 2^{31} \overset{u}{<} y + 2^{31}$$

$$x \overset{u}{<} y \;=\; x - 2^{31} < y - 2^{31}$$

类似的关系对$<$、$\overset{u}{<}$等也成立。加2^{31}和减2^{31}等价，因为它们相当于颠倒符号位。

从无符号比较得到带符号比较的另一个方法是基于下面的事实: 如果x和y的符号相同，那么$x<y=x\overset{u}{<}y$，而如果x和y的符号不同，那么$x<y=x\overset{u}{>}y$ [Lamp]。同样，其逆变换也正确，所有我们有:

$$x < y \;=\; (x \overset{u}{<} y) \oplus x_{31} \oplus y_{31} \quad \text{和}$$

23

$$x \overset{u}{<} y \;=\; (x < y) \oplus x_{31} \oplus y_{31}$$

其中，x_{31}和y_{31}分别是x和y的符号位。类似的关系对$<$、$\overset{u}{<}$等也成立。

在大多数计算机上，=和≠以外的所有常用比较谓词都可以使用这些比较谓词中的任何一个和最多3个额外的指令来计算。例如，我们把$x\overset{u}{<}y$作为基本比较谓词，因为这个谓词是最容

易实现的一个谓词（它是$y-x$的进位位）。那么，如下所示，可以得到其他谓词：

$$x < y = \neg(y + 2^{31} \overset{u}{\leqslant} x + 2^{31})$$

$$x \leqslant y = x + 2^{31} \overset{u}{\leqslant} y + 2^{31}$$

$$x > y = \neg(x + 2^{31} \overset{u}{\leqslant} y + 2^{31})$$

$$x \geqslant y = y + 2^{31} \overset{u}{\leqslant} x + 2^{31}$$

$$x \overset{u}{<} y = \neg(y \overset{u}{\leqslant} x)$$

$$x \overset{u}{>} y = \neg(x \overset{u}{\leqslant} y)$$

$$x \overset{u}{\geqslant} y = y \overset{u}{\leqslant} x$$

1. 从进位位得到比较谓词

如果计算机能够很容易地把进位位传送到通用寄存器中，那么就可以对某些比较谓词做简明的编码。下面列出这些关系中的几个。标记carry(*expression*)表示表达式中最外层操作所产生的进位位。我们假设减$x-y$的进位位是由对$x+\bar{y}+1$的加法器产生的，它是“借位”的补。

$x = y$: carry($0 - (x - y)$)，或 carry($(x + y) + 1$)，或 carry($(x - y - 1) + 1$)

$x \neq y$: carry($(x - y) - 1$)，即，carry($(x - y) + (-1)$)

$x < y$: $\neg$carry($(x + 2^{31}) - (y + 2^{31})$)

$x \leqslant y$: carry($(y + 2^{31}) - (x + 2^{31})$)

$x \overset{u}{<} y$: $\neg$carry($x - y$)

$x \overset{u}{\leqslant} y$: carry($y - x$)

$x = 0$: carry($0 - x$)，或 carry($\bar{x} + 1$)

$x \neq 0$: carry($x - 1$)，即，carry($x + (-1)$)

$x < 0$: carry($x + x$)

$x \leqslant 0$: carry($2^{31} - (x + 2^{31})$)

对于$x>y$，我们使用$x \leqslant y$的表达式的补，其他包括“大于”比较的关系也是类似的。 24

在IBM RS/6000和它的近亲PowerPC上，可以将GNU超级优化器运用于谓词表达式的计算问题[GK]。RS/6000有关于abs(x)、nabs(x)、doz(x, y)以及若干使用进位位的*add*和*subtract*形式的指令。我们发现，对于所有的整数比较谓词，RS/6000最多使用3个（1周期）基本指令就能计算，这一结果甚至令计算机的设计者都感到吃惊。这里的“所有谓词”包括6个二元操作数带符号比较和4个二元操作数无符号比较，以及第二个操作数为0的相应比较，所有比较产生1/0值或-1/0值结果。PowerPC没有abs(x)、nabs(x)和doz(x, y)指令，它可以用最多4个基本指令计算所有的比较谓词。

2. 计算机如何设置比较谓词

大多数计算机都具有计算整数比较谓词得到1位结果的方法。某些计算机将这个结果位放入“条件寄存器”，而有些计算机（如我们的RISC模型）则将结果位放入通用寄存器。无论哪种情况，通常都是这样来实现上述操作的：对比较操作数做减法，然后在结果位上执行少

量逻辑操作来决定1位比较结果。

下面是关于这些操作的理论。假设计算机通过计算$x+\bar{y}+1$来计算$x-y$，而且可以在结果中获得下述量：

C_o，高阶位位置出来的进位

C_i，进入高阶位位置的进位

N，结果的符号位

Z，若除C_o外结果的各相关量都是0则等于1，否则等于0

于是，我们有下列布尔代数表记的比较操作（并置表示与，+表示或）：

V:	$C_i \oplus C_o$	（带符号溢出）
$x = y$:	Z	
$x \neq y$:	$\bar{Z}$	
$x < y$:	$N \oplus V$	
$x \leqslant y$:	$(N \oplus V) + Z$	
$x > y$:	$(N \equiv V)\bar{Z}$	
$x \geqslant y$:	$N \equiv V$	
$x \overset{u}{<} y$:	$\overline{C_o}$	
$x \overset{u}{\leqslant} y$:	$\overline{C_o} + Z$	
$x \overset{u}{>} y$:	$C_o\bar{Z}$	
$x \overset{u}{\geqslant} y$:	C_o	

25

2.12 溢出检测

“溢出”的意思就是算术操作的结果太大或太小，不能用目标寄存器正确表示。本节讨论如何不使用通常为此目的而专设的“状态位”来检测溢出是否发生。这是非常重要的，因为有些计算机没有这样的状态位（例如，MIPS计算机），而且还因为即使计算机有这样的状态位，高级语言通常也很难甚至不可能访问这些位。

1. 带符号加、减法

当加或减发生溢出时，现代计算机无一例外地忽略结果的高阶位，而存储加法器自然生成的低阶位。当且仅当操作数符号相同而且和的符号与操作数的符号相反时，带符号整数加法发生溢出。令人惊讶的是，即使有输入到加法器的进位，这条规则同样成立。也就是说，当计算的是$x+y+1$时，这条规则同样成立。这对多字带符号整数加法的运用是非常重要的，在这种加法中，后一步的加运算是两个全字和一个进位的带符号加，进位可能是0或+1。

为了证明加法的这条规则，设x、y表示要做加法运算的一个字长的带符号整数的值，设c（进位）是0或1。同时，为简单起见，假设使用的是一台4位计算机。那么，如果x和y的符号不同，则有

$$-8 \leqslant x \leqslant -1, \text{ 且}$$
$$0 \leqslant y \leqslant 7$$

如果x是非负数，y是负数，则有与上述边界公式类似的边界公式。无论哪种情况，将这些不等式加起来，并考虑c而额外加1，有

$$-8 \leqslant x+y+c \leqslant 7$$

这可以用一个4位带符号整数表示，因此当操作数符号不同时，不会发生溢出。

现在，假设x和y的符号相同。这时有两种情况：

(a)	(b)
$-8 \leqslant x \leqslant -1$	$0 \leqslant x \leqslant 7$
$-8 \leqslant y \leqslant -1$	$0 \leqslant y \leqslant 7$

因此，

(a)	(b)
$-16 \leqslant x+y+c \leqslant -1$	$0 \leqslant x+y+c \leqslant 15$

26

当不能用4位带符号整数表示和时就会发生溢出，也就是说，如果

(a)	(b)
$-16 \leqslant x+y+c \leqslant -9$	$8 \leqslant x+y+c \leqslant 15.$

时会发生溢出。对于(a)的情况，这等价于4位和的最高阶位为0，与x和y的符号相反。对于(b)的情况，这等价于4位和的最高阶位为1，同样与x和y的符号相反。

对于多字整数的减，重要的计算是$x-y-c$，其中c同样是**0**或**1**，**1**表示借位。从与上类似的分析中可以看到，当且仅当x和y符号相反且$x-y-c$的符号与x相反时（或等价地说与y的符号相同），$x-y-c$的值发生溢出。

这导致下面的溢出谓词表达式，它的结果在符号位给出。在其后加入右移位31个位或带符号右移位31个位分别生成1/0值和-1/0值结果。

$x+y+c$	$x-y-c$
$(x \equiv y) \,\&\, ((x+y+c) \oplus x)$	$(x \oplus y) \,\&\, ((x-y-c) \oplus x)$
$((x+y+c) \oplus x) \,\&\, ((x+y+c) \oplus y)$	$((x-y-c) \oplus x) \,\&\, ((x-y-c) \equiv y)$

通过选择第一栏中的第二个选择和第二栏中的第一个选择（避免等值操作），除所需的$x+y+c$和$x-y-c$的计算外，我们的基本RISC计算机可以用额外3个指令来进行这些检测。也可以加上第4个指令分支到处理溢出条件的代码上（负分支指令）。

由于溢出而产生执行中断时，程序员也许希望用一种不引发中断的方法去检测某个加法或减法是否会产生溢出。下面是一个无分支的不中断检测方法：

$x+y+c$	$x-y-c$
$z \leftarrow (x \equiv y) \,\&\, \mathbf{0x80000000}$	$z \leftarrow (x \oplus y) \,\&\, \mathbf{0x80000000}$
$(x \equiv y) \,\&\, ((x \oplus z)+y+c) \equiv y$	$(x \oplus y) \,\&\, ((x \oplus z)-y-c) \oplus y$

在左边一栏中，如果x和y的符号相同则z的赋值为$z=$**0x8000 0000**，如果x和y的符号不同则z的赋值为$z=$**0**。然后，在第二个表达式中，对符号不同的x和y来进行加运算，所以不产生溢出。如果x和y是非负数，第二个表达式中的符号位的值是1当且仅当$(x-2^{31})+y+c \geqslant 0$，即当且仅当$x+y+c \geqslant 2^{31}$；这正是计算$x+y+c$时溢出发生的条件。如果$x$和$y$是负的，那么第二个表达式中的符号位的值是1当且仅当$(x+2^{31})+y+c<0$，即当且仅当$x+y+c<-2^{31}$；这同样是溢出发生的条件。

27

如果x和y的符号相反，项$x \equiv y$确保结果正确（在符号位置上是0）。类似说明也适用于减法的情况（右移）。在基本RISC计算机中，这一代码的执行需要9个指令。

如果加法所生成的进位可用的话，那么它似乎对计算带符号溢出谓词有帮助。情况并非如此。然而，这一思路导致了下面的方法。

如果x是一个带符号整数，那么$x+2^{31}$可以正确地表示一个无符号数，而且可以通过颠倒x的高阶位得到这个无符号数。如果$x+y \geq 2^{31}$，也就是说$(x+2^{31}) + (y+2^{31}) \geq 3 \times 2^{31}$，那么将在正方向发生带符号溢出。条件$(x+2^{31}) + (y+2^{31}) \geq 3 \times 2^{31}$可以用无符号加法的进位（这表示和大于或等于$2^{32}$）以及和的高阶位为1来刻画。类似地，当进位是0而且和的高阶位也是0时，在负方向发生溢出。

这样，我们有如下检测带符号加法的溢出算法：

计算$(x \oplus 2^{31})+(y \oplus 2^{31})$、给出和$s$和进位$c$。

当且仅当c等于s的高阶位时发生溢出。

对带符号加法来说，和是正确的，因为颠倒两个操作数的高阶位不改变它们的和。

对于减法，除了第一步用减法代替加法之外，算法相同。我们假设通过计算$x+\bar{y}+1$来计算$x-y$产生的进位。对带符号减法，结果是正确的差。

这些公式也许有趣，但是在大多数计算机上，它们并不比不使用进位的公式更有效。（例如，加法的溢出$=(x \equiv y)\&(s \oplus x)$，减法的溢出$=(x \oplus y)\&(d \oplus x)$，其中，$s$和$d$分别表示$x$和$y$的和与差。）

2. 计算机如何对带符号加、减法设置溢出

计算机通常通过“进符号位置的进位与出符号位置的进位不相等”的理论方法对带符号加法设置“溢出”。有意思的是，假设用$x+\bar{y}+1$来计算$x-y$，这种理论能够同时给出正确的加法和减法溢出检测。此外，无论是进位还是借位，它都是正确的。在软件中，这似乎并不能产生计算带符号溢出谓词的好方法，然而，虽然如此，它能够轻易地把进位存放到符号位中。对于加法和减法，通过计算下面的表达式，可以从符号位得到进位或借位（其中c是0或1）：

进位	借位
$(x+y+c) \oplus x \oplus y$	$(x-y-c) \oplus x \oplus y$

28 事实上，对每个位置i，这些表达式给出进入位置i的进位和借位。

3. 无符号加、减法

下面的无分支代码可以用来计算无符号加/减法的溢出谓词，结果放入符号位。涉及右移位的这些表达式可能只有当我们知道c=0时有用。方括号中的表达式计算最小有效位生成的进位或借位。

$x+y+c$，无符号的

$$(x \,\&\, y) \mid ((x \mid y) \,\&\, \neg(x+y+c))$$

$$(x \overset{u}{\gg} 1) + (y \overset{u}{\gg} 1) + [((x \,\&\, y) \mid ((x \mid y) \,\&\, c)) \,\&\, 1]$$

$x-y-c$，无符号的

$$(\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x-y-c))$$

$$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y - c))$$

$$(x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - [((\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, c)) \,\&\, 1]$$

对于无符号的加和减，有比使用比较操作简单得多的公式[MIPS]。对于无符号加法，如果和（通过无符号比较）小于任何一个操作数，那么就发生溢出（进位）。下面给出这一溢出判定公式和类似的公式。不幸的是，在这些公式中没有用变量c来表示进位或借位的方法。而程序必须检测c，根据c是0或1使用不同类型的比较。

$x+y$，无符号的	$x+y+1$，无符号的	$x-y$，无符号的	$x-y-1$，无符号的
$\neg x \overset{u}{<} y$	$\neg x \overset{u}{\leqslant} y$	$x \overset{u}{<} y$	$x \overset{u}{\leqslant} y$
$x+y \overset{u}{<} x$	$x+y+1 \overset{u}{\leqslant} x$	$x-y \overset{u}{>} x$	$x-y-1 \overset{u}{\geqslant} x$

上面每种情况的第一个公式是在可能发生溢出的加法或减法之前计算的，这样就给出了不产生溢出的检测方法。在可能发生溢出的加法或减法之后，计算每种情况的第二个公式。

似乎没有这样简单的（使用比较）计算带符号溢出谓词的方法。

4. 乘法

对于乘法，溢出意味着结果不能用32位表示（结果总可以用64位表示，无论是带符号还是无符号）。如果可以访问积的高阶32位，那么很容易检测溢出。我们用hi($x \times y$)和lo($x \times y$)分别表示64位积的高阶32位和低阶32位。那么，可以如下计算溢出谓词[MIPS]: 29

$x \times y$，无符号的	$x \times y$，带符号的
$\mathrm{hi}(x \times y) \neq 0$	$\mathrm{hi}(x \times y) \neq (\mathrm{lo}(x \times y) \overset{s}{\gg} 31)$

检测乘法溢出的方法之一就是先做乘积，然后用除运算来检测结果。但是必须注意：不能除以0，而且带符号乘法会更加复杂。如果下面的表达式为**真**则发生溢出：

无符号的	带符号的
$z \leftarrow x * y$	$z \leftarrow x * y$
$y \neq 0 \;\vec{\&}\; z \overset{u}{\div} y \neq x$	$(y < 0 \,\&\, x = -2^{31}) \mid (y \neq 0 \;\vec{\&}\; z \div y \neq x)$

当$x=-2^{31}$且$y=-1$时，情况变得比较复杂。在这种情况下可能发生乘法溢出，但计算机给出-2^{31}的结果。这将引发除法的溢出，因此（对某些计算机）任何结果都可能出现。所以，必须单独检查这种情况，项$y<0 \,\&\, x=-2^{31}$可以完成这一任务。为了防止除以0，上面的表达式使用了“条件与”操作符（在C语言中，使用&&操作符）。

也可以不做乘法（也就是说、不引发溢出），用除法来检查乘法溢出。对于无符号整数，当且仅当$xy>2^{32}-1$或$x>((2^{32}-1)/y)$时乘积溢出；或者，因为x是整数，当$x>\lfloor(2^{32}-1)/y\rfloor$时乘积溢出。用计算机算术表示，就是：

$$y \neq 0 \;\vec{\&}\; x \overset{u}{>} (\mathbf{0xFFFFFFFF} \overset{u}{\div} y)$$

对于带符号整数，确定$x*y$是否溢出不是那样简单。如果x和y的符号相同，那么当且仅当$xy>2^{31}-1$时发生溢出。如果它们符号不同，那么当且仅当$xy<-2^{31}$时发生溢出。这些条件可用表2-2所示的方法来检测，这些检测利用带符号除法。

表2-2 带符号乘法的溢出检测

	$y>0$	$y\leqslant 0$
$x>0$	$x>\mathbf{0x7FFFFFFF}\div y$	$y<\mathbf{0x80000000}\div x$
$x\leqslant 0$	$x<\mathbf{0x80000000}+y$	$x\neq 0\ \vec{\&}\ y<\mathbf{0x7FFFFFFF}\div x$

30

由于有四种情况，实现这些检测比较难。因为溢出的问题以及不能表示数$+2^{31}$，因此很难把这些表达式统一起来。

如果可以使用无符号除法，则能够简化这一检测。我们可以使用x和y的绝对值，而这一绝对值在无符号整数的解释之下，可以正确地表示。下面是一个完整的检测算法。如果x和y的符号相同，则变量$c=2^{31}-1$，否则$c=2^{31}$。

$$c\leftarrow((x\equiv y)\overset{s}{\gg}31)+2^{31}$$
$$x\leftarrow \mathrm{abs}(x)$$
$$y\leftarrow \mathrm{abs}(y)$$
$$y\neq 0\ \vec{\&}\ x\overset{u}{>}(c\overset{u}{\div}y)$$

前导零数目指令可以用来评估$x*y$是否会产生溢出，再将这一评估进一步求精，从而精确给出是否产生溢出的判断。首先，考虑无符号整数的乘法。很容易知道，如果作为32位量的x和y分别有m和n个前导零，那么64位积有$m+n$或$m+n+1$个前导零（如果$x=0$或$y=0$则有64个前导零）。如果64位积的前导零数目少于32，那么发生溢出。因此，有

$\mathrm{nlz}(x)+\mathrm{nlz}(y)\geqslant 32$：乘法一定不发生溢出。

$\mathrm{nlz}(x)+\mathrm{nlz}(y)<30$：乘法一定发生溢出。

当$\mathrm{nlz}(x)+\mathrm{nlz}(y)=31$时，可能发生、也可能不发生溢出。在这种情况下，可以通过计算$t=x\lfloor y/2\rfloor$来做溢出的评估。它不会产生溢出。因为y是偶数时，$xy=2t$，或当y是奇数时$xy=2t+x$，如果$t\geqslant 2^{31}$，那么积xy产生溢出。这种考虑导致以下计算xy的方案：当发生溢出时分支转换到“溢出”。这个方案如图2-2所示。

```
unsigned x, y, z, m, n, t;

m = nlz(x);
n = nlz(y);
if (m + n <= 30) goto overflow;
t = x*(y >> 1);
if ((int)t < 0) goto overflow;
z = t*2;
if (y & 1) {
   z = z + x;
   if (z < x) goto overflow;
}
// z is the correct product of x and y.
```

图2-2 无符号乘法溢出的判断

对于带符号整数乘法，根据非负自变量的前导零数目和负自变量前导1数目，我们可以对

是否产生溢出给出部分判断。设

$$m=\mathrm{nlz}(x)+\mathrm{nlz}(\bar{x}),\ \text{且}$$

$$n=\mathrm{nlz}(y)+\mathrm{nlz}(\bar{y})$$

那么，我们有

$m+n>34$：乘法一定不发生溢出。

$m+n<31$：乘法一定发生溢出。

31

有两个不明确的情况：$m+n=32$和$m+n=33$。对于$m+n=33$的情况，只有当两个自变量都是负数，而且真实的积为2^{31}（计算机结果是-2^{31}）时发生溢出，所以，可以通过检测积的符号是否正确来判断溢出（也就是说，如果$m\oplus n\oplus(m*n)<0$则发生溢出）。当$m+n=32$时，很难区分发生溢出还是不发生溢出。

对此我们不再做进一步的探讨，只是注意：也可以基于$\mathrm{nlz}(\mathrm{abs}(x)+\mathrm{nlz}(\mathrm{abs}(y))$来评估带符号乘法的溢出，但是同样有两种不明确的情况（$\mathrm{nlz}(\mathrm{abs}(x))+\mathrm{nlz}(\mathrm{abs}(y))=31$和$\mathrm{nlz}(\mathrm{abs}(x))+\mathrm{nlz}(\mathrm{abs}(y))=32$时）。

5. 除法

对于带符号除法$x\div y$，如果下面的表达式为**真**，则发生溢出：

$$y=0\mid(x=\mathbf{0x80000000}\ \&\ y=-1)$$

大多数计算机对不确定的$0\div 0$发出溢出（或中断）信号。

评估这个表达式的直接代码，包括最后到溢出处理代码的分支，由7个指令组成，其中3个是分支。似乎没有什么更好的技巧来改进这些，但是下面是可能的改进：

$$[\mathrm{abs}(y\oplus\mathbf{0x80000000})\mid(\mathrm{abs}(x)\ \&\ \mathrm{abs}(y\equiv\mathbf{0x80000000}))]<0$$

也就是说，评估方括号中的长的表达式，如果结果小于0则分支。在拥有上面所指定的指令且“与0比较”不需代价的计算机上，这一计算需要约9个指令，包括常量的装入和最后的分支。

另外的可能是，首先根据如下表达式计算z：

$$z\leftarrow(x\oplus\mathbf{0x80000000})\mid(y+1)$$

32

（在许多计算机上需要3个指令），然后用下面所给出的方法之一做溢出检测并对$y=0\mid z=0$做分支：

$$((y\mid -y)\ \&\ (z\mid -z))\geqslant 0$$

$$(\mathrm{nabs}(y)\ \&\ \mathrm{nabs}(z))\geqslant 0$$

$$((\mathrm{nlz}(y)\mid\mathrm{nlz}(z))\overset{u}{\gg}5)\neq 0$$

在拥有所指定指令的计算机上，这些方法分别需要9个、7个和8个指令。最后一个表达式对PowerPC机来说是一个好方法。

对于无符号除法$x\overset{u}{\div}y$，当且仅当$y=0$时发生溢出。

2.13　加、减、乘的特征码结果

许多计算机提供“特征码”（condition code）来刻画整数算术操作的结果。通常，只有一

个加指令，而它的表示特征反映出我们是把操作数解释为带符号数还是无符号数（不能是混合类型），同时反映出运算的结果。这一表示特征常常由下列成分组成：

- 是否产生进位（无符号溢出）。
- 是否有带符号溢出。
- 当32位运算结果被解释为带符号2的补码整数，而且忽略进位和溢出时，它是负、0还是正。

一些老式计算机对无限精确结果（也就是说，加和减的33位结果）是正、负还是0给出指示。然而，高级语言的编译器不太容易使用这一指示，因此它不再受欢迎。

对于加法，这些事件的12种组合中只有9种可能发生。不可能发生的事件组合有“无进位、溢出、结果>0”、“无进位、溢出、结果=0”、“进位、溢出、结果<0”。因此，特征码正好需要4位。这些组合中有两个是惟一的，即，在输入值中只有一个值能产生相应的情况：只有0加0具有“无进位、无溢出、结果=0”的特征，只有最大负数加最大负数具有“进位、溢出、结果=0”的特征。如果有“进位”，也就是说，如果我们计算的是$x+y+1$的话，那么上面的陈述仍然是正确的。

对于减法，我们假设计算$x-y$时，计算机实际上计算的是$x+\bar{y}+1$，进位的产生与加指令的情况相同（在这一方案中，“进位”的意思被颠倒，即，进位=1意味着结果能放入一个字，进位=0意味着结果不能放入一个字）。于是，对于减法，上述的12种组合中只有7种可能发生。不可能发生的情况除了加法中不可能发生的组合之外，再加上“无进位、无溢出、结果=0”和“进位、溢出、结果=0”。

33

如果计算机的乘法器能够生成双字结果，那么最好有两个乘指令：一个是对无符号操作数做乘法，另一个是对带符号操作数做乘法。（在一台4位计算机上，十六进制数的带符号乘F × F的结果是01，无符号乘F × F的结果是E1。）对于这些指令，在结果总能装入双字的意义下，进位和溢出都不会发生。

对于生成一个字结果（双字结果的低位字）的乘法指令，我们认为“进位”意指当操作数和结果都是无符号整数时结果不能放入一个字，而“溢出”意指当操作数和结果被解释成带符号2的补码整数时结果不能放入一个字。那么，同样有9种组合可能发生，不可能发生的是“无进位、溢出、结果>0”、“无进位、溢出、结果=0”和“进位、无溢出、结果=0”。因此，同时考虑加、减和乘指令时，有10种组合可能发生。

2.14 循环移位

这些公式是一目了然的。可能令人惊讶的是，这一代码对从0到32的n均正确，即使移位是模32时也是如此。

$$\text{左循环移位}n\text{个位：}\ y \leftarrow (x \ll n) \mid (x \overset{u}{\gg} (32-n))$$
$$\text{右循环移位}n\text{个位：}\ y \leftarrow (x \overset{u}{\gg} n) \mid (x \ll (32-n))$$

2.15 双字长加、减法

使用前面2.12节中所给出的无符号加法和减法的溢出表达式，可以不访问计算机的进位位而轻松地实现双字长加法和减法。下面以双字长加法为例进行说明。设操作数是(x_1, x_0)和$(y_1,$

y_0)，结果是(z_1，z_0)。下标1表示最高有效字，下标0表示最低有效字。假设使用寄存器的所有32个位。最低有效字是无符号量。

$$z_0 \leftarrow x_0 + y_0$$

$$c \leftarrow [(x_0 \,\&\, y_0) \mid ((x_0 \mid y_0) \,\&\, \neg z_0)] \overset{u}{\gg} 31$$

$$z_1 \leftarrow x_1 + y_1 + c$$

这一代码的执行需要9个指令。第二行也可以是$c \leftarrow (z_0 \overset{u}{<} x_0)$，在一台有相应比较操作符并把结果1或0放入寄存器的计算机上，例如，MIPS计算机中的“SLTU”(Set on Less Than Unsigned)指令，可以有4指令解。 34

双字长减法($x-y$)的代码类似:

$$z_0 \leftarrow x_0 - y_0$$

$$b \leftarrow [(\neg x_0 \,\&\, y_0) \mid ((x_0 \equiv y_0) \,\&\, z_0)] \overset{u}{\gg} 31$$

$$z_1 \leftarrow x_1 - y_1 - b$$

在一台有完整逻辑指令集的计算机上，执行以上代码需要8个指令。第二条也可以是$b \leftarrow (x_0 \overset{u}{<} y_0)$，在一台有“SLTU”指令的计算机上，有4指令解。

在大多数计算机上，通过只使用最低有效字的31位来表示多字长数据，可以用5个指令来完成双字长加法和减法。最低有效字的最高阶位除临时存放进位或借位外，其余时候均为0。

2.16　双字长移位

设(x_1，x_0)是32位字序偶，并要对它进行左或右移位，把它们当作一个64位量，x_1表示最高有效字。设(y_1，y_0)是操作结果，解释与上相同。假设移位量n的变化范围是0到63。进一步假设计算机的移位指令是模64或更大的。也就是说，除非移位是带符号右移位，移位量在32到63或-32到-1时将导致一个所有位均为0的字，而对于带符号右移位，该移位的结果是一个32个位都来自被移位字的符号位的字。(在Intel x86上，这个代码不能正确工作，这种计算机拥有的是模32移位)。

在上述的假设下，双字长左移位操作可如下实现（8个指令):

$$y_1 \leftarrow x_1 \ll n \mid x_0 \overset{u}{\gg} (32-n) \mid x_0 \ll (n-32)$$

$$y_0 \leftarrow x_0 \ll n$$

在第一个赋值中，为了给出当n=32时也正确的结果，主要连结词必须是或而不能是加。如果已知$0 < n < 32$，则可以忽略第一个赋值中的最后一项，因而给出6指令解。

类似地，双字长无符号右移位操作可以用下面的式子来实现。

$$y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32-n) \mid x_1 \overset{u}{\gg} (n-32)$$

$$y_1 \leftarrow x_1 \overset{u}{\gg} n$$ 35

双字长带符号右移位更加困难，因为其中有一项存在一个多余的符号传播。简明的代码如下:

$$\text{if } n < 32 \text{ then } y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32-n)$$

$$\text{else } y_0 \leftarrow x_1 \overset{s}{\gg} (n-32)$$

$$y_1 \leftarrow x_1 \overset{s}{\gg} n$$

如果计算机中有条件传送指令，那么很容易把这一代码表示成无分支代码，它需要8个指令。如果不能使用条件传送指令，那么这个操作也可以这样实现：用带符号右移位31指令构建一个掩码来屏蔽掉这个多余的符号传播项：

$$y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32-n) \mid [(x_1 \overset{s}{\gg} (n-32)) \,\&\, ((32-n) \overset{s}{\gg} 31)]$$

$$y_1 \leftarrow x_1 \overset{s}{\gg} n$$

2.17 多字节加、减、绝对值

有些应用处理短整数数组（通常是字节或半字），如果每次对一个字进行操作，那么处理的速度会更快。为明确起见，下述例子处理将四个1字节整数封装到一个字的情况，但是，这些技术很容易改编以适应其他封装形式，例如，在一个字中封装一个12位整数和两个10位整数，等等。这些方法在64位计算机上非常有用，因为这样可以并行地完成更多的工作。

加运算必须在能够阻挡从一个字节到另一字节的进位的方式下完成。用下面给出的两步方法可以实现这种加运算：

1）屏蔽掉每个操作数的每个字节的高阶位并执行加指令（这样就不会产生越过字节边界的进位）。

2）对两个操作数的高阶位和进入到高阶位的进位做1位加，以此来修正每个字节的高阶位。

当然，进到每个字节的高阶位的进位是第一步计算的和的高阶位。对于减法也有类似的方法：

加法

$$s \leftarrow (x \,\&\, \mathbf{0x7F7F7F7F}) + (y \,\&\, \mathbf{0x7F7F7F7F})$$

$$s \leftarrow ((x \oplus y) \,\&\, \mathbf{0x80808080}) \oplus s$$

减法

$$d \leftarrow (x \mid \mathbf{0x80808080}) - (y \,\&\, \mathbf{0x7F7F7F7F})$$

$$d \leftarrow ((x \oplus y) \mid \mathbf{0x7F7F7F7F}) \equiv d$$

36

在一台有完整逻辑指令集的计算机上，这些运算的执行需要8个指令，其中包括**0x7F7F 7F7F**的装入。（分别将**0x80808080**的与和或改为**0x7F7F7F7F**的与非和或非。）

当一个字只被分成两个字段时，存在一个不同的方法。在这种情况下，可以用32位加再跟随一个减去多余进位的操作来实现加运算。在2.12节提到过，表达式$(x+y)\oplus x\oplus y$给出了进到每一个位置的进位。运用这个表达式并对减法做类似的考察，得到如下所示的关于模2^{16}加/减两个半字的代码（需要7个指令）：

加法	减法
$s \leftarrow x + y$	$d \leftarrow x - y$
$c \leftarrow (s \oplus x \oplus y) \,\&\, \mathbf{0x00010000}$	$b \leftarrow (d \oplus x \oplus y) \,\&\, \mathbf{0x00010000}$
$s \leftarrow s - c$	$d \leftarrow d + b$

多字节绝对值指令也很容易实现，只需求补并把1加到每个存放负整数的字节（它们的高阶位为1）。下面的代码把*y*的每个字节设置成*x*的相应字节的绝对值（8个指令）：

```
a←x & 0x80808080          //析出符号
b←a >>u 7                 //x中的字节存放的是负数时，b的相应字节为整数1
m←(a-b) | a               //x中的字节存放的是负数时，m的相应字节为0xFF
y←(x⊕m) + b               //对x中存放负数的字节取补并加1
```

第三行也可写成$m \leftarrow a+a-b$。第四行中的加*b*不能越过字节边界进位，因为$x \oplus m$在每个字的高阶位都是0。

2.18 doz、max、min函数

“doz”函数是“差或零”，对带符号参数，其定义如下：

$$\mathrm{doz}(x,y) = \begin{cases} x-y, & x \geqslant y \\ 0, & x < y \end{cases}$$

它曾被称为“一年级减法”，因为如果试图取走太多的话，则结果为0。我们将用这个函数来实现max(*x*, *y*)和min(*x*, *y*)。为此，我们要特别注意doz(*x*, *y*)可以取负值；当减法产生溢出时它就是负的。可以直接使用差或零函数实现Fortran的IDIM函数，尽管在Fortran中，如果有溢出发生，结果一般是无定义的。

37

在实现doz(*x*, *y*)、max(*x*, *y*)和min(*x*, *y*)上，不存在既适用于大多数计算机又无分支的好方法。我们所能够做的最好的方法就是，用2.11节所给$x<y$谓词的一个表达式来计算doz(*x*, *y*)，再由此计算max(*x*, *y*)和min(*x*, *y*)，如下所示：

$$d \leftarrow x - y$$

$$\mathrm{doz}(x,y) = d \,\&\, [(d \equiv ((x \oplus y) \,\&\, (d \oplus x))) \overset{s}{\gg} 31]$$

$$\max(x,y) = y + \mathrm{doz}(x,y)$$

$$\min(x,y) = x - \mathrm{doz}(x,y)$$

如果计算机有等值指令，计算doz(*x*, *y*)则需要7个指令，否则需要8个指令；而计算max(*x*, *y*)或min(*x*, *y*)需要再加一个指令。

下面是这些函数的无符号版本：

$$d \leftarrow x - y$$

$$\mathrm{dozu}(x,y) = d \,\&\, \neg[((\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, d)) \overset{s}{\gg} 31]$$

$$\mathrm{maxu}(x,y) = y + \mathrm{dozu}(x,y)$$

$$\mathrm{minu}(x,y) = x - \mathrm{dozu}(x,y)$$

在IBM RISC/6000计算机以及它的前任801计算机上，有单独的doz(*x*, *y*)指令。这样，我们可以用2个指令计算带符号整数的max(*x*, *y*)和min(*x*, *y*)，而且有时它自身也非常有用。直接实现max(*x*, *y*)和min(*x*, *y*)的代价更高，因为计算机需要一个从寄存器文卷的输出口到输入口的通道来绕过ALU。

具有条件传送的计算机可以用2个指令实现得到破坏性的[⊖] max(x, y)和min(x, y)。例如，在完全RISC计算机上，可以按下面方法计算$x \leftarrow \max(x, y)$（我们把目标寄存器放在最前面）：

```
cmplt   z,x,y          Set z = 1 if x < y, else 0.
movne   x,z,y          If z is nonzero, set x = y.
```

2.19 交换寄存器

有一种不使用第三个寄存器而交换两个寄存器的内容的古老技巧[IBM]：

$$x \leftarrow x \oplus y$$
$$y \leftarrow y \oplus x$$
38
$$x \leftarrow x \oplus y$$

这一操作在二地址的计算机上工作得很好。用逻辑操作≡（异或的补）取代⊕同样可以实现这一技巧。同样，也可以用带有加和减的各种方法来实现这一技巧：

$x \leftarrow x + y$	$x \leftarrow x - y$	$x \leftarrow y - x$
$y \leftarrow x - y$	$y \leftarrow y + x$	$y \leftarrow y - x$
$x \leftarrow x - y$	$x \leftarrow y - x$	$x \leftarrow x + y$

不幸的是，这些方法都有一个不适于二地址计算机的指令，除非计算机有“反向减”。

这个小技巧可以实际运用于双缓冲的应用中，用于交换指针。第一条指令可以置于交换的循环之外（虽然这一操作抹煞了节省寄存器的优势）：

在循环外部：$t \leftarrow x \oplus y$
在循环内部：$x \leftarrow x \oplus t$
$y \leftarrow y \oplus t$

1. 交换寄存器的相应字段

这里的问题是：对于两个寄存器x和y以及掩码m，当第i位的掩码m_i=1时，交换x和y的第i位内容；而当第i位的掩码m_i=0时，保留x和y的第i位的内容不变。这里，“相应”字段指的是不需要移位。m的1位不必是连续的。直接的方法如下：

$$x' \leftarrow (x \,\&\, \bar{m}) \mid (y \,\&\, m)$$
$$y \leftarrow (y \,\&\, \bar{m}) \mid (x \,\&\, m)$$
$$x \leftarrow x'$$

通过使用“临时寄存器”来存放4个与表达式的值，假设装入m 和$\bar{m}$都只需1个指令，而且计算机有单指令与非，那么这一方法需要7个指令。如果计算机能够并行执行4个（独立的）与表达式，那么执行时间只有3个周期。

下面(a)栏所给的也许是一个更好的方法（需要5个指令，但是在无限制指令级并行的计算机上需要4个周期）。它通过“三异或”代码实现寄存器的交换。

⊖ 破坏性操作是改写它的一个或多个参数的操作。

(a)	(b)	(c)
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$t \leftarrow (x \oplus y)\ \&\ m$
$y \leftarrow y \oplus (x\ \&\ m)$	$y \leftarrow y \equiv (x \mid \bar{m})$	$x \leftarrow x \oplus t$
$x \leftarrow x \oplus y$	$x \leftarrow x \equiv y$	$y \leftarrow y \oplus t$

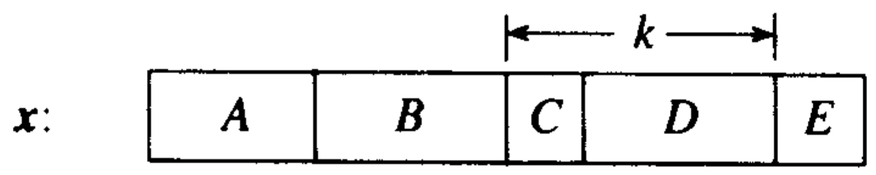
39

(b)栏与(a)栏做同样的交换，但是当m不能放入一个立即字段，而$\bar{m}$能放入一个立即字段，而且计算机有等值指令时，(b)栏非常有用。

上面的(c)栏给出了另外一个方法[GLS1]。它也需要5个指令（同样假设必须使用一个指令来把m装入寄存器），但是在具有足够的指令级并行的计算机上执行只需要3个周期。

2. 同一寄存器的两个字段的交换

假设一个寄存器x有两个（长度相同的）字段需要交换，而其他位不发生变化。也就是说，目标是对于下图所示的计算字交换字段B和D，而不改变字段A、C和E。字段由移位距离k分隔。



简明的代码将把D和B移位到它们的新位置上，并用与和或操作把这些字组合起来，如下所示：

$$t_1 = (x\ \&\ m) \ll k$$

$$t_2 = (x \overset{u}{\gg} k)\ \&\ m$$

$$x' = (x\ \&\ m') \mid t_1 \mid t_2$$

这里，m是对应于字段D的各位的值为1的掩码（在其他位置为0），m'是对应于字段A、C、E的各位的值为1的掩码。在具有无限制指令级并行的计算机上，这个代码需要9个指令和4个周期，包括2个装入掩码m和m'的指令。

在同样的假设之下，下面是一个只需要7个指令执行时间为5个周期的方法[GLS1]。它与上文(c)栏的交换两个寄存器的相应字段的代码相似。同样，m是析出字段D的掩码。

$$t_1 = [x \oplus (x \overset{u}{\gg} k)]\ \&\ m$$

$$t_2 = t_1 \ll k$$

$$x' = x \oplus t_1 \oplus t_2$$

这里的思路是，在位置D上t_1包含$B \oplus D$，在位置B上t_2包含$B \oplus D$。当B和D是“分字段”（split field），也就是说，掩码m的1位不连续时，这一代码以及前面给出的简明代码也能够正确运行。

40

3. 条件交换

当掩码m为0时，前两小节基于异或的交换方法退化为无操作。因此，这些方法可以根据如果某个条件c为**真**时则设置m为所有位均为1，如果c为**假**时设置m为所有位均为0的掩码，来分别对整个寄存器、两个寄存器的相应字段或一个寄存器中的两个字段进行交换。如果能够无分支地设置m，那么以上给出无分支代码。

2.20 两个或更多值之间的交换

假设一个变量x只能取两个值a和b，而你希望给x赋值为与它当前值不同的值，而且还希望代码独立于a和b的值。例如，在一个编译器中，x可能是已知分支真或分支假的操作码，而且不管它是什么，你想把它转换到另一个。操作码分支真或分支假的值是任给的，也许是在C语言的头文件中用#define或enum来定义的。

做转换的简明代码是：

```
if (x == a) x = b;
else x = a;
```

或C程序中常见的

```
x = x == a ? b : a;
```

一个好得多（至少更有效）的编码方法是

$$x \leftarrow a + b - x, \text{ 或者}$$
$$x \leftarrow a \oplus b \oplus x$$

如果a和b是常量，它们只需要1到2个基本RISC指令。当然，可以忽略$a + b$中的溢出。

这引发一个问题：有在三个或更多的值之间循环的特别有效的方法吗？也就是说，任给三个互不相同的常量a、b和c，我们要寻找一个容易计算的函数f，使得它满足：

$$f(a) = b$$
$$f(b) = c$$
$$f(c) = a$$

41

有意思的是，这样的函数总存在多项式形式。对于三个常量的情况，函数的多项式形式为：

$$f(x) = \frac{(x-a)(x-b)}{(c-a)(c-b)}a + \frac{(x-b)(x-c)}{(a-b)(a-c)}b + \frac{(x-c)(x-a)}{(b-c)(b-a)}c \tag{2-4}$$

（思路是：如果$x=a$，则第一项和最后一项消失，中间的项简化为b，以此类推。）它的计算需要14个算术操作，而且，对任给的a、b和c，中间结果可能超出计算机的字长。但是，它只是一个二次方程式；如果把它写成多项式的普通形式，并用Horner规则[⊖]计算的话，将只需要5个算术操作（对整系数二次多项式需要4个算术操作，最后的除需要1个算术操作）。重新排列等式(2-4)可得：

$$\begin{aligned} f(x) = \frac{1}{(a-b)(a-c)(b-c)}\{&[(a-b)a + (b-c)b + (c-a)c]x^2 \\ &+ [(a-b)b^2 + (b-c)c^2 + (c-a)a^2]x \\ &+ [(a-b)a^2b + (b-c)b^2c + (c-a)ac^2]\} \end{aligned}$$

这个多项式变得太复杂，没有意义（没有实用性）。

另外一个方法如下。对每个常量只有一个项残存下来，这一点与等式(2-4)类似。

$$f(x) = ((-(x = c)) \,\&\, a) + ((-(x = a)) \,\&\, b) + ((-(x = b)) \,\&\, c)$$

⊖ Horner规则就是简单地把x分解出来。例如，它把四次多项式$ax^4+bx^3+cx^2+dx+e$变换成$x(x(x(ax+b)+c)+d)+e$来计算。对于n次多项式它做n个乘运算和n个加运算，非常适合于乘-加指令。

如果计算机具有相等谓词，那么这个多项式的计算需要11个指令（不包括常量的装入）。因为有2个加操作要把0与一个非零值结合，我们可以用或或异或操作来代替它们。

可以先行计算$a-c$和$b-c$来简化上述公式，然后再使用[GLS1]的方法：

$$f(x) = ((-(x = c)) \,\&\, (a - c)) + ((-(x = a)) \,\&\, (b - c)) + c, \quad 或$$

$$f(x) = ((-(x = c)) \,\&\, (a \oplus c)) \oplus ((-(x = a)) \,\&\, (b \oplus c)) \oplus c$$

42

这两个操作都需要8个指令。但是在大多数计算机上，它们可能没有如下所示的C语言的简明代码好。对于小常量a、b和c，它需要4到6个指令。

```
if (x == a) x = b;
else if (x == b) x = c;
else x = a;
```

继续探讨这一问题，有一个巧妙的适用于没有比较谓词指令计算机的方法，即在三个常量间循环的无分支方法[GLS1]。在大多数计算机上，它的执行需要8个指令。

因为a、b和c互不相同，存在两个位的位置n_1和n_2，在这两个位置上a、b和c的相应位不全相等，而且在这两个位置上的“单数出局”（odd one out）是不同的（单数指的是在这个位置上的值与另两个不同）。下面是a、b和c分别为21、31和20时的示例，用二进制表示。

```
1 0 1 0 1    c
1 1 1 1 1    a
1 0 1 0 0    b
  n1    n2
```

不失一般性，重新命名a、b和c，使得a在位置n_1上单数出局，b在位置n_2上单数出局，如上所示。那么在位置n_1上的位值组合有两种可能，即$(a_{n_1}, b_{n_1}, c_{n_1}) = (0, 1, 1)$或$(1, 0, 0)$。类似地，在位置$n_2$上的位值组合也有两种可能，即$(a_{n_2}, b_{n_2}, c_{n_2}) = (0, 1, 0)$或$(1, 0, 1)$。总共有四种情况，对应于每种情况的公式如下：

情况1　$(a_{n_1}, b_{n_1}, c_{n_1}) = (0, 1, 1), (a_{n_2}, b_{n_2}, c_{n_2}) = (0, 1, 0)$:

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (c - a) + b$$

情况2　$(a_{n_1}, b_{n_1}, c_{n_1}) = (0, 1, 1), (a_{n_2}, b_{n_2}, c_{n_2}) = (1, 0, 1)$:

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (a - c) + (b + c - a)$$

情况3　$(a_{n_1}, b_{n_1}, c_{n_1}) = (1, 0, 0), (a_{n_2}, b_{n_2}, c_{n_2}) = (0, 1, 0)$:

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (c - a) + a$$

情况4　$(a_{n_1}, b_{n_1}, c_{n_1}) = (1, 0, 0), (a_{n_2}, b_{n_2}, c_{n_2}) = (1, 0, 1)$:

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (a - c) + c$$

43

在这些公式中，每一个乘法的左操作数都是一个位。乘0或1可以分别转换成与各位为0或各位为1的数的与。因此，第一个公式可以改写成如下公式（其余公式也可以做同样的改写）：

$$f(x) = ((x \ll (31-n_1)) \overset{s}{\gg} 31)\&(a-b) + ((x \ll (31-n_2)) \overset{s}{\gg} 31)\&(c-a) + b$$

因为除x之外所有变量都是常量，在基本RISC计算机中用8个指令就可以计算这一公式。同样，加和减可以用异或取代。

这一思路可以扩展到4个或更多常量之间的循环。这一思路的精髓是寻找位的位置n_1、n_2、…，在这些位置上的位值可以惟一地区分这些常量。对于4个常量，最多需要3个位的位置。于是，（对于4个常量）解下面关于s、t、u和v的等式（也就是，解由4个线性等式构成的联立方程组，其中$f(x)$是a、b、c或d，系数x_n是0或1）：

$$f(x) = x_{n_1}s + x_{n_2}t + x_{n_3}u + v$$

如果4个常量只用两个位的位置就可以惟一区分的话，那么需要解的等式就是：

$$f(x) = x_{n_1}s + x_{n_2}t + x_{n_1}x_{n_2}u + v$$

44

第3章　2的幂边界

3.1　上舍入、下舍入到已知的2的幂的倍数

很容易将无符号整数x下舍入成如下所示的下一个较小的8的倍数：$x\&-8$就可以完成这一工作。另一个选择是$(x \overset{u}{>>} 3)<<3$。如果“下舍入”意指向负方向舍入，那么这些方法对带符号整数也同样适用（例如，$(-37)\&(-8)=-40$）。

上舍入也很简单。例如，可以用下面任何一个公式把无符号整数x上舍入到下一个较大的8的倍数。

$$(x+7)\&-8，或$$

$$x+(-x\&7)$$

如果“上舍入”意指向正方向舍入，那么这些表达式同样适用于带符号整数。可以利用第二个表达式的第二项来看要在x上加上多少才能使它成为8的倍数[Gold]。

显然，我们可以把上面的两个表达式结合在一起，把带符号整数向0的方向舍去到最近的8的倍数：

$$t \leftarrow (x \overset{s}{>>} 31)\ \&\ 7;$$

$$(x+t)\ \&\ -8$$

如果计算机没有立即与，或式中的常量相对于其立即字段来说太大的话，可以把第一行改成$t \leftarrow (x \overset{s}{>>} 2) \overset{u}{>>} 29$。

有时，舍入的因子是以调整量的$\log_2$的形式给出的（例如，值3意味着舍入到8的倍数）。在这种情况之下，可以利用下面的代码，其中，$k = \log_2$（调整量）：

下舍入：
$$x\ \&\ ((-1) \ll k)$$

$$(x \overset{u}{>>} k) \ll k$$

上舍入：
$$t \leftarrow (1 \ll k) - 1;\ \ (x+t)\ \&\ \neg t$$

$$t \leftarrow (-1) \ll k;\ \ (x-t-1)\ \&\ t$$

3.2　上舍入、下舍入到下一个2的幂

我们定义两个与“地板”函数和“天花板”函数相似的函数，但是它们都直接舍入到最近的2的整数幂，而不是最近的整数。数学上，它们被定义为：

$$\mathrm{flp2}(x) = \begin{cases} 未定义, & x<0, \\ 0, & x=0, \\ 2^{\lfloor \log_2 x \rfloor}, & x>0 \end{cases} \qquad \mathrm{clp2}(x) = \begin{cases} 未定义, & x<0, \\ 0, & x=0, \\ 2^{\lceil \log_2 x \rceil}, & x>0 \end{cases}$$

函数名的第一个字母分别意指“地板”函数和“天花板”函数。因此，flp2(x)是小于等于x的

最大的2的幂，而clp2(x)是大于等于x的最小的2的幂。即使x不是整数，这些定义也有意义（例如，flp2(0.1)=0.0625）。这些函数满足与“地板”函数和“天花板”函数类似的关系，下面是其中的一些关系，其中n是一个整数。

$$\lfloor x \rfloor = \lceil x \rceil \text{ 当且仅当}x\text{是一个整数} \qquad \text{flp2}(x) = \text{clp2}(x) \text{ 当且仅当}x\text{是2的幂或0}$$
$$\lfloor x + n \rfloor = \lfloor x \rfloor + n \qquad \text{flp2}(2^n x) = 2^n \text{flp2}(x)$$
$$\lceil x \rceil = -\lfloor -x \rfloor \qquad \text{clp2}(x) = 1/\text{flp2}(1/x),\ x \neq 0$$

从计算的角度，我们只研究x是整数时的情况，而且只考虑无符号整数，这样，函数对所有的x都有定义。我们要计算的值是算术上的正确值模2^{32}（也就是说，对于$x>2^{31}$，我们取clp2(x)为0）。下面是对若干x的函数值列表：

x	flp2(x)	clp2(x)
0	0	0
1	1	1
2	2	2
3	2	4
4	4	4
5	4	8
…	…	…
$2^{31}-1$	2^{30}	2^{31}
2^{31}	2^{31}	2^{31}
$2^{31}+1$	2^{31}	0
…	…	…
$2^{32}-1$	2^{31}	0

46

函数flp2和 clp2间有如下关系。通过这些关系，可以使用一个函数来计算另一个函数，条件是满足所给的限制。

$$\begin{aligned}
\text{clp2}(x) &= 2\,\text{flp2}(x-1), && x \neq 1 \\
&= \text{flp2}(2x-1), && 1 \leqslant x \leqslant 2^{31} \\
\text{flp2}(x) &= \text{clp2}(x \mathbin{\overset{u}{\div}} 2 + 1), && x \neq 0 \\
&= \text{clp2}(x+1) \mathbin{\overset{u}{\div}} 2, && x < 2^{31}
\end{aligned}$$

如下所示，使用前导零数目指令很容易计算上舍入函数和下舍入函数。然而，为了使这些关系对x=0和$x>2^{31}$成立，计算机必须拥有当移位量为−1、32和63时生成0的移位指令。很多计算机（例如，PowerPC）就有“模64”移位，它可以完成这一工作。对于−1的情况，只要计算机能够向相反方向移位即可（也就是说，左移位−1位变成右移位1位）。

$$\begin{aligned}
\text{flp2}(x) &= 1 \ll (31 - \text{nlz}(x)) \\
&= 1 \ll (\text{nlz}(x) \oplus 31) \\
&= 0\text{x}80000000 \mathbin{\overset{u}{\gg}} \text{nlz}(x) \\
\text{clp2}(x) &= 1 \ll (32 - \text{nlz}(x-1)) \\
&= 0\text{x}80000000 \mathbin{\overset{u}{\gg}} (\text{nlz}(x-1) - 1)
\end{aligned}$$

1. 下舍入

图3-1描述了在前导零数目指令不可用时可采用的无分支算法。这个算法基于右传播最左侧的1位，它的执行需要12个指令。

图3-2给出了两个计算下舍入函数的简单循环算法。所有变量都是无符号整数。右侧的循环持续把x的最右侧的1位变为0，直到$x=0$为止，然后返回先前的x值。

```
unsigned flp2(unsigned x) {
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return x - (x >> 1);
}
```

图3-1 计算不大于x的最大的2的幂，无分支

47

```
   y = 0x80000000;
   while (y > x)
      y = y >> 1;
   return
```

```
do {
   y = x;
   x = x & (x - 1);
} while(x != 0);
return y;
```

图3-2 计算不大于x的最大的2的幂，简单循环

左侧的循环的执行需要4nlz(x)+3个指令。当$x \neq 0$时，如果与0的比较不需代价，那么右侧的循环的执行需要4pop(x)个指令[⊖]。

2. 上舍入

右传播技巧可以为计算上舍入到下一个2的幂提供好算法。图3-3所示的算法无分支，并可用12个指令来执行。

```
unsigned clp2(unsigned x) {
   x = x - 1;
   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return x + 1;
}
```

图3-3 不小于x的最小的2的幂

使用简单循环不能很好地计算上舍入：

```
y = 1;

while (y < x)          // 无符号比较。
```

⊖ pop(x)是x中1位的数目。

```
    y = 2*y;
return y;
```

对于$x=0$，这一代码返回1，这也许不是你想要的。当$x \geq 2^{31}$时代码会陷入无限循环。对于其他的x，执行需要$4n+3$个指令，其中n是返回整数的2的幂。因此，从执行的指令数目看，对$n \geq 3(x \geq 8)$，它比无分支代码要慢。

48

3.3 检测2的幂的边界跨越

假设从地址0开始，内存被分成大小为2的某个幂的块。这些块可以是字、双字、页等等。然后，给定一个开始地址a和一个长度l，希望确定从a到$a+l-1$(其中$l \geq 2$)的地址区间是否跨越块的边界。a和l可以是任何能够放入寄存器的无符号量。

如果$l=0$或1，那么无论a是什么都不会发生边界跨越。如果l超出了块的大小，那么无论a是什么都一定会发生边界跨越。对于非常大的l（允许绕回），即使地址区间的第一个和最后一个字节在同一个块中，也可能发生边界跨越。

IBM System/370有一个检测边界跨越的极其简明的方法[CJS]。下面是块的大小为4096字节（普通页长）时的这一检测方法。

```
O    RA,=A(-4096)
ALR  RA,RL
BO   CROSSES
```

第一个指令计算RA（它包含开始地址a）和0xFFFF F000的逻辑或。第二个指令把长度加进来，同时设置计算机的2位特征码。如果产生进位，则这一逻辑加指令将特征码的第一位设置为1；如果32位寄存器中的结果非零，那么它将特征码的第二位设置为1。如果特征码的两个位都是1，那么最后一条指令进行分支。在分支目标上，RA将含有超出第一页的长度（这是一个没被要求的额外特性）。

例如，如果$a=0$和$l=4096$，那么发生进位，而寄存器中的结果是0，因此这个程序正确地不分支到标签CROSSES。

让我们来看一看如何使这一方法适用于RISC计算机，RISC计算机通常没有进位及寄存器结果非零分支指令。为了简明起见，这里使用大小为8的块，那么如果发生进位（$(a|-8)+l \geq 2^{32}$）而且寄存器中的结果不为零（$(a|-8)+l \neq 2^{32}$），那么[CJS]的方法分支到CROSSES。因此，这与下面的谓词等值。

$$(a \mid -8) + l > 2^{32}$$

这又等价于在计算$((a|-8)-1)+l$的最后一步加中获得进位。如果计算机有进位分支指令，可以直接使用这一公式，这将给出大约5个指令的解，包括常量-8的装入。

如果计算机没有进位分支指令，我们可以利用当且仅当$\neg x <^{u} y$时$x+y$发生进位的事实（参见2.12节的“无符号加、减法”）来得到表达式：

49

$$\neg((a \mid -8) - 1) <^{u} l$$

使用各种恒等式，例如$\neg(x-1)=-x$，可以给出如下相互等值的“边界跨越”谓词的表达式：

$$-(a \mid -8) <^{u} l$$

$$\neg(a \mid -8) + 1 <^{u} l$$

$$(\neg a \,\&\, 7) + 1 \overset{u}{<} l$$

在大多数RISC计算机上，这些表达式大约要用5到6个指令来计算。

下面是另一种方案。显然，对于8字节边界，当且仅当

$$(a \,\&\, 7) + l - 1 \geqslant 8$$

时发生边界跨越。由于有溢出的可能（当l非常大时发生溢出），这一公式是不能直接计算的。但是，上式很容易重新排列为$8-(a\&7)<l$，它可以在计算机上直接计算（它的任何部分都不发生溢出）。这样，我们得到下面的表达式：

$$8 - (a \,\&\, 7) \overset{u}{<} l$$

在大多数RISC计算机上，它可以用5个指令计算（如果有从立即值减指令则可以用4个指令计算）。如果发生边界跨越，超出第一块的长度可以用$l-(8-(a\&7))$给出，可以使用一个额外的减指令来计算它。

50

第4章　算术边界

4.1　整数的边界检测

所谓的“边界检测”就是检测一个整数x是否在两个边界a和b之内，也就是说：

$$a \leqslant x \leqslant b$$

我们首先假设所有量都是带符号整数。

边界检测的一个重要应用是检测数组下标。例如，假设一个一维数组A可以用1到10做下标。那么，对于引用$A(i)$，编译器可能生成检测

$$1 \leqslant i \leqslant 10$$

而且当上式不成立时进行分支或中断。本节指出，只用一个比较指令就能完成这种检测，那就是执行一个与其等值的检测[PL8]：

$$i - 1 \overset{u}{<} 9$$

这也许是更好的代码，因为它只包含一个比较分支（或比较中断）指令，而且也许我们本来就需要$i-1$来计算数组元素的地址。

那么，即使在减运算中发生溢出时，

$$a \leqslant x \leqslant b \Rightarrow x - a \overset{u}{\leqslant} b - a$$

也总是正确的吗？是的，只要我们知道$a \leqslant b$，那么它总是正确的。对于数组边界检测，语言规则可能会要求数组中取0或负数的元素（或沿着任何轴的取0或负的元素）不超过一定数目。可以在编译的时候检验这一规则，或者在分配数组的时候动态地检验这一规则。下面将证明，在这样的环境下，上面的转换是正确的。

下面的引理给我们的证明带来方便，它本身也很有价值。

引理　*如果a和b是带符号整数且$a \leqslant b$，那么当计算值$b-a$被解释成无符号整数时，$b-a$正确地代表$b-a$的算术值。*

证明（假设计算机是32位的）　因为$a \leqslant b$，真差$b-a$的取值范围是0到$(2^{31}-1)-(-2^{31})=2^{32}-1$。 51

如果真差的范围在0到$2^{31}-1$，那么计算机的结果是正确的（因为这一结果在带符号解释下可表示），而且符号位为0。因此，无论是带符号解释还是无符号解释，计算机的结果都是正确的。

如果真差的范围在2^{31}到$2^{32}-1$之间，那么计算机结果与真差相差2^{32}的某个倍数（因为结果在带符号解释下不可表示）。这导致结果（在带符号解释下）在-2^{31}到-1之间。计算机结果比真差小2^{32}，符号位为1。重新把这一结果解释为无符号数，那么因为符号位的权是$+2^{31}$而不是-2^{31}，这使结果增加2^{32}。因此，重新解释的结果是正确的。

下面是“边界定理”。

定理 若a和b为带符号整数且$a<b$，则有

$$a \leqslant x \leqslant b \;=\; \boldsymbol{x-a \overset{u}{\leqslant} b-a} \tag{4-1}$$

证明 根据x的值，我们分三种情况证明。在所有情况中，因为$a<b$，根据引理，如果$\boldsymbol{b-a}$被解释为无符号值，那么$\boldsymbol{b-a}$的计算值等于算术值$b-a$，如等式(4-1)所示。

情况1，$x<a$：在这种情况下，$\boldsymbol{x-a}$的无符号解释是$x-a+2^{32}$。（在32位数的范围）无论x和b的值是多少都有

$$x + 2^{32} > b$$

因此

$$x - a + 2^{32} > b - a$$

所以

$$\boldsymbol{x-a \overset{u}{>} b-a}$$

在这种情况下，等式(4-1)的两边都为**假**。

情况2，$a<x<b$：这时有算术表达式$x-a<b-a$。因为$a<x$，根据引理，如果$\boldsymbol{x-a}$的值被解释为无符号数，那么$x-a$等于$\boldsymbol{x-a}$的计算值。因此，

$$\boldsymbol{x-a \overset{u}{\leqslant} b-a}$$

也就是说，等式(4-1)的两边都为**真**。

情况3，$x>b$：这时有$x-a>b-a$。因为在这种情况下$x>a$（因为$b>a$），根据引理，如果$\boldsymbol{x-a}$的值被解释为无符号数，那么$x-a$等于$\boldsymbol{x-a}$的值。因此

$$\boldsymbol{x-a \overset{u}{>} b-a}$$

52 也就是说，等式(4-1)的两边都为**假**。

当a和b是无符号整数时，上述定理依然为真。这是因为对于无符号整数，引理显然成立，而且上述证明也是显然的。

下面是由上述定理得来的一组边界检测的变形。这些变形对a、b和x的带符号或无符号解释都成立。

$$\begin{aligned}
&\text{if } a \leqslant b \text{ then } a \leqslant x \leqslant b \;=\; \boldsymbol{x-a \overset{u}{\leqslant} b-a \;=\; b-x \overset{u}{\leqslant} b-a}\\
&\text{if } a \leqslant b \text{ then } a \leqslant x < b \;=\; \boldsymbol{x-a \overset{u}{<} b-a}\\
&\text{if } a \leqslant b \text{ then } a < x \leqslant b \;=\; \boldsymbol{b-x \overset{u}{<} b-a}\\
&\text{if } a < b \text{ then } a < x < b \;=\; \boldsymbol{x-a-1 \overset{u}{<} b-a-1 \;=\; b-x-1 \overset{u}{<} b-a-1}
\end{aligned} \tag{4-2}$$

在最后一条规则中，可以用$\boldsymbol{b+\neg a}$代替$\boldsymbol{b-a-1}$。

当检测的类型是$-2^{n-1} \leqslant x < 2^{n-1}-1$时，一些完全不同的变形也许更有用。这一检测判断带符号量x是否能正确地表示成n位的2的补码整数。以n=8为例进行说明，下面的检测是等价的：

a. $\boldsymbol{-128 \leqslant x \leqslant 127}$

b. $\boldsymbol{x + 128 \overset{u}{\leqslant} 255}$

c. $\boldsymbol{(x \overset{s}{\gg} 7) + 1 \overset{u}{\leqslant} 1}$

d. $x \overset{s}{\gg} 7 = x \overset{s}{\gg} 31$

e. $(x \overset{s}{\gg} 7) + (x \overset{u}{\gg} 31) = 0$

f. $(x \ll 24) \overset{s}{\gg} 24 = x$

g. $x \oplus (x \overset{s}{\gg} 31) \leqslant 127$

等式(b)是本节前面所述内容的简单应用。将x右移位7个位后，等式(c)也是如此。等式(c)到等式(f)以及(g)也许仅在等式(a)和等式(b)中的常量超过计算机的比较和加指令的立即字段的长度时有用。

与2的幂有关的另一个特殊情况是:

$$0 \leqslant x \leqslant 2^n - 1 \Leftrightarrow (x \overset{u}{\gg} n) = 0$$

或更一般地，

$$a \leqslant x \leqslant a + 2^n - 1 \Leftrightarrow ((x-a) \overset{u}{\gg} n) = 0$$ 53

4.2 通过加和减传播边界

一些优化的编译器进行表达式的“值域分析”。这是确定程序中出现的表达式的上下边界的步骤。尽管这种优化并不能真正发挥很大的作用，但是它的确允许我们改进编译器，诸如在C语言的“开关”语句中加入原本省略了的值域检测，以及加入原本省略了的下标边界检测，从而为程序调试提供帮助。

假设我们有两个变量x和y，它们的边界如下，其中所有量都是无符号的:

$$\begin{aligned} a \leqslant x \leqslant b, \text{ 且} \\ c \leqslant y \leqslant d \end{aligned} \tag{4-3}$$

那么，我们如何计算$x+y$、$x-y$以及$-x$的紧界（tight bound）呢？当然，在算术上有$a+c \leqslant x+y \leqslant b+d$；但是关键是加法可能发生溢出。

计算边界的方法如下:

定理 *若a、b、c、d、x及y为无符号整数，并且有*

$$a \overset{u}{\leqslant} x \overset{u}{\leqslant} b \text{ 且}$$

$$c \overset{u}{\leqslant} y \overset{u}{\leqslant} d,$$

则:

$$\begin{aligned} &\text{若} a+c < 2^{32}-1 \text{且} b+d > 2^{32}, \text{ 则} 0 \overset{u}{\leqslant} x+y \overset{u}{\leqslant} 2^{32}-1, \\ &\quad \text{否则 } a+c \overset{u}{\leqslant} x+y \overset{u}{\leqslant} b+d \end{aligned} \tag{4-4}$$

$$\begin{aligned} &\text{若} a-d<0 \text{且} b-c \geqslant 0, \text{ 则} 0 \overset{u}{\leqslant} x-y \overset{u}{\leqslant} 2^{32}-1, \\ &\quad \text{否则 } a-d \overset{u}{\leqslant} x-y \overset{u}{\leqslant} b-c \end{aligned} \tag{4-5}$$

$$\begin{aligned} &\text{若} a=0 \text{且} b \neq 0, \text{ 则} 0 \overset{u}{\leqslant} -x \overset{u}{\leqslant} 2^{32}-1, \\ &\quad \text{否则} -b \overset{u}{\leqslant} -x \overset{u}{\leqslant} -a \end{aligned} \tag{4-6}$$

不等式(4-4)表明$x+y$的边界“一般”是$a+c$和$b+d$，但是如果$a+c$的计算不发生溢出而$b+d$ 54

的计算发生溢出时，边界是0和最大无符号整数。可以对等式(4-5)做类似解释，但是当减法的真结果小于0时（在负方向上）产生溢出。

证明 如果$a+c$和$b+d$都不产生溢出，那么对指定值域内的x和y，$x+y$不产生溢出，这使计算结果与真正结果相等，因此式(4-4)的第二个不等式成立。如果$a+c$和$b+d$都产生溢出，那么$x+y$也产生溢出。现在，算术上很显然有

$$a + c - 2^{32} \leqslant x + y - 2^{32} \leqslant b + d - 2^{32}$$

然而，这正是当三项都溢出时的计算公式。因此在这种情况下也有

$$a + c \overset{u}{\leqslant} x + y \overset{u}{\leqslant} b + d$$

如果$a+c$不产生溢出，但$b+d$产生溢出，那么有

$$a + c < 2^{32} - 1 \quad \text{且} \quad b + d \geqslant 2^{32}$$

因为$x+y$的取值介在$a+c$与$b+d$之间，它的取值也在$2^{32}-1$与2^{32}之间，也就是说，$x+y$的计算值的取值在$2^{32}-1$与0之间（尽管它不一定能取那个区间内的所有值）。

最后，对于$a+c$产生溢出但$b+d$不产生溢出的情况，等式成立的理由是$a < b$且$c < d$。

以上是不等式(4-4)的完整证明。式(4-5)的证明类似，但是“溢出”意味着真差小于0。

不等式(4-6)可以通过式(4-5)和$a=b=0$以及变量换名证明。（对于无符号整数x，表达式$-x$指的是计算$2^{32}-x$的值，如果愿意也可以认为这是计算$\neg x+1$的值。）

因为无符号溢出很容易判断（参见2.12节的“无符号加、减法”），这些结果很容易嵌入到代码中，加法和减法的代码嵌入见图4-1。其中，计算出的下界和上界分别是变量s和t。

```
s = a + c;
t = b + d;
if (s >= a && t < b) {
   s = 0;
   t = 0xFFFFFFFF;}
```

```
s = a - d;
t = b - c;
if (s > a && t <= b) {
   s = 0;
   t = 0xFFFFFFFF;}
```

55

图4-1 在加法和减法操作中传播无符号边界

带符号数

带符号数的边界检测不是那么清晰。同上，假设我们有两个变量x和y，它们的边界如下，其中所有量都是带符号数：

$$a \leqslant x \leqslant b, \text{ 且}$$
$$c \leqslant y \leqslant d \tag{4-7}$$

我们希望计算$x+y$、$x-y$和$-x$的紧界。理由与无符号数的边界检测非常相似，对于加法，结果如下：

$$\begin{aligned}
a + c < -2^{31}, b + d < -2^{31} &: \boldsymbol{a + c \leqslant x + y \leqslant b + d} \\
a + c < -2^{31}, b + d \geqslant -2^{31} &: \boldsymbol{-2^{31} \leqslant x + y \leqslant 2^{31} - 1} \\
-2^{31} \leqslant a + c < 2^{31}, b + d < 2^{31} &: \boldsymbol{a + c \leqslant x + y \leqslant b + d} \\
-2^{31} \leqslant a + c < 2^{31}, b + d \geqslant 2^{31} &: \boldsymbol{-2^{31} \leqslant x + y \leqslant 2^{31} - 1} \\
a + c \geqslant 2^{31}, b + d \geqslant 2^{31} &: \boldsymbol{a + c \leqslant x + y \leqslant b + d}
\end{aligned} \tag{4-8}$$

第一行表明，如果$a+c$和$b+d$在负的方向上产生溢出，那么$x+y$的计算和就位于$a+c$与$b+d$

的计算和之间。这是因为所有三个计算和都超过了相同的量（2^{32}）。第二行表明，当$a+c$在负的方向产生溢出，而$b+d$或者不产生溢出或者在正的方向产生溢出时，$x+y$的计算和取值于最小负数与最大正数之间（尽管也许不能取其间的所有值），这不难证明。对其他行也可以做类似的解释。

通过重写y的边界为

$$-d \leqslant -y \leqslant -c$$

并且利用加操作的边界传播规则，我们很容易得到减操作对带符号数边界的传播规则。这些结果如下所示：

$$\begin{aligned}
a-d<-2^{31}, b-c<-2^{31} &: \boldsymbol{a-d \leqslant x-y \leqslant b-c} \\
a-d<-2^{31}, b-c\geqslant-2^{31} &: \boldsymbol{-2^{31} \leqslant x-y \leqslant 2^{31}-1} \\
-2^{31}\leqslant a-d<2^{31}, b-c<2^{31} &: \boldsymbol{a-d \leqslant x-y \leqslant b-c} \\
-2^{31}\leqslant a-d<2^{31}, b-c\geqslant 2^{31} &: \boldsymbol{-2^{31} \leqslant x-y \leqslant 2^{31}-1} \\
a-d\geqslant 2^{31}, b-c\geqslant 2^{31} &: \boldsymbol{a-d \leqslant x-y \leqslant b-c}
\end{aligned} \quad \boxed{56}$$

取负操作的边界传播规则可以通过在减操作的规则中取$a=b=0$、省略某些不可能出现的组合并简化、变量换名而得到。结果如下：

$$\begin{aligned}
a=-2^{31}, b=-2^{31} &: \boldsymbol{-x=-2^{31}} \\
a=-2^{31}, b\neq-2^{31} &: \boldsymbol{-2^{31} \leqslant -x \leqslant 2^{31}-1} \\
a\neq-2^{31} &: \boldsymbol{-b \leqslant -x \leqslant -a}
\end{aligned}$$

带符号数边界检测的C语言代码很杂乱，我们只考虑加法的边界传播操作。似乎最简单的办法就是去检测式(4-8)中边界的计算值取极小负数与极大正数的两种情况。如果两个操作数是负的、而和是非负的，那么就在负的方向产生溢出（参见2.12节中的“带符号加、减法”）。因此，为了检测条件$a+c<-2^{31}$，我们可以设`s = a + c`；然后用类似于“`if(a<0&&c<0&&s>=0)...`”这样的语句编码。然而，直接对算术变量实施逻辑操作并将逻辑操作的真/假值结果放入变量的符号位会更有效[⊖]。这样，我们把上面的条件写成“`if((a&c&~s)<0)...`”。综上所述，我们得到图4-2所示的程序片断。

```
s = a + c;
t = b + d;
u = a & c & ~s & ~(b & d & ~t);
v = ((a ^ c) | ~(a ^ s)) & (~b & ~d & t);
if ((u | v) < 0) {
    s = 0x80000000;
    t = 0x7FFFFFFF;}
```

图4-2 加法操作的边界传播

在这里，当加运算a+c在负的方向产生溢出，而且加运算b+d在负的方向不产生溢出时，u为**真**（符号位为1）。如果加运算a+c在正的方向不产生溢出，而且加运算b+d在正的方向产

⊖ 这里，更有效指的是代码更紧凑、分支少；为了得到运行速度更快的代码，我们可以假定边界过大的可能性较少，从而首先对无溢出的情况进行检测。

生溢出，那么变量v为**真**。前一个条件可以表述为“a和c的符号不同，或a和s符号相同”。图
57 中的if检测与“if(u<0 ||v<0)等价——也就是说，如果u或v为**真**”。

4.3 逻辑操作的边界传播

与上节相同，假设我们有两个变量x和y，它们的边界如下，其中所有量都是无符号的：

$$a \leqslant x \leqslant b，且$$
$$c \leqslant y \leqslant d \tag{4-9}$$

那么，$x \mid y$、$x\&y$、$x \oplus y$以及$\neg x$的紧界是什么呢？

将不等式(4-9)与2.3节的不等式相结合，并注意到$\neg x = 2^{32} - 1 - x$，我们得到

$$\max(a, c) \leqslant (x \mid y) \leqslant b + d$$
$$0 \leqslant (x \,\&\, y) \leqslant \min(b, d)$$
$$0 \leqslant (x \oplus y) \leqslant b + d$$
$$\neg b \leqslant \neg x \leqslant \neg a$$

其中，我们假设加运算$b+d$没有溢出。对于上节我们提到的在编译器中的应用，这些表达式更容易计算，也许更好。然而，前两个不等式中的边界不是紧界。例如，用二进制表示常量，假设：

$$\mathbf{00010} \leqslant x \leqslant \mathbf{00100}，且$$
$$\mathbf{01001} \leqslant y \leqslant \mathbf{10100} \tag{4-10}$$

那么，通过观察（例如，试一试x和y的所有36种可能性），我们可以看到$\mathbf{01010} < (x \mid y) < \mathbf{10111}$。因此，下界不是$\max(a, c)$，也不是$a \mid c$，同样上界不是$b+d$，也不是$b \mid d$。

给定不等式(4-9)中的a、b、c和d的值，怎样得到这些逻辑表达式的紧界呢？首先考虑由$x \mid y$达到的最小值。合理的猜测可能是当x和y达到最小时的表达式的值，也就是$a \mid c$。然而，式(4-10)所给的例子表明，最小值可能比$a \mid c$小。

为了找到这个最小值，我们的步骤是，从$x=a$和$y=c$开始，寻找这样一个增量：用它来增加x或y的值时$x \mid y$的值会减小。结果就是这个减小了的$x \mid y$的值。我们直接使用a和c而不是把它们分别赋值给x和y，然后在下面的条件下增加a和c中的一个：增加后的值在合理的范围内并且$a \mid c$的值减小。

这一步骤是，从左到右仔细扫描a和c的每个位。如果二者的相应位都是0，那么结果的相应位也是0。如果二者的相应位都是1，那么结果的相应位也是1（显然，没有x和y的值可以使结果变小）。对于这两种情况，继续扫描下一位。如果被扫描的位中一个是1而另一个是0，那么就把这个0改成1，并且把其后的所有位都设置成0，这样可以减小$a \mid c$的值。因为在这一位
58 上另一个边界为1，$a \mid c$在这一位置上为1，所以这种改变不会增加$a \mid c$的值。因此，这一步骤构成一个在此位置上的0被变成1、而后面的位置上的值都变成0的数。如果它小于或等于相应的上界，那么这种改变就是可行的；做这一改变，结果就是这一修正值与另一个下界的或。如果这种改变不可行（原因是这一改变了的值超过了相应的上界），那么继续扫描下一位。

这就是整个步骤。似乎进行这种修改之后，可以继续扫描，寻找其他减小$a \mid c$的值的机会。然而，即使我们找到一个位置，它允许这一位上的0变成1，而且可以把其后的所有位都设置成0，也不能减小$a \mid c$的值，因为你所重置的那个下界的那些位已经都是0了。

对应于这一算法的C语言代码如图4-3所示。我们假设编译器将把子表达式~a&c和a&~c移到循环外。更重要的是，如果前导零数目指令可用，那么这个程序可以通过使用下面的式子对m初始化来加速。

```
m = 0x80000000 >> nlz(a ^ c);
```

这一代码将跳过a和c同为0或1的初始位位置。当a^c为0（即a=c）时，为了使这种提速产生效果，计算机的右移位指令应该是模64的。如果前导零数目指令不可用，那么也许值得对参数a^c使用flp2函数的某些变形（参见3.2节）。

```
unsigned minOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & c & m) {
         temp = (a | m) & -m;
         if (temp <= b) {a = temp; break;}
      }
      else if (a & ~c & m) {
         temp = (c | m) & -m;
         if (temp <= d) {c = temp; break;}
      }
      m = m >> 1;
   }
   return a | c;
}
```

图4-3 相对于x和y的边界的$x \mid y$的最小值

59

现在我们来考虑$x \mid y$的最大值问题，变量x和y的边界如不等式(4-9)所示。算法与最小值的算法相似，只不过它要（从左到右）扫描边界b和d的值，寻找二者都为1的位置。如果找到了这样的位置，算法通过把其中一个边界的相应位置上的1变成0，并把它后面的所有位都设置成1来增加$c \mid d$的值。如果这是可行的（如果结果的值大于或等于相应的下界），那么这种改变是可行的，从而得到修改后的边界值。如果这种改变不可行，那么就在另一个边界上试一试。如果两个边界都不可行，继续往下扫描。此算法的C语言代码如图4-4所示。其中，子表达式b&d可以移到循环之外，通过下面的式子初始化m，可以提高算法的速度。

```
m = 0x80000000 >> nlz(b & d);
```

对于不等式(4-9)，我们有两种传播表达式$x \,\&\, y$的边界的方法：代数方法和直接计算方法。代数方法使用DeMorgan律：

$$x \,\&\, y = \neg(\neg x \mid \neg y)$$

因为我们知道传播或的边界的精确方法，以及非的边界传播方法（即，$a \overset{u}{<} x \overset{u}{<} b \Leftrightarrow \neg b \overset{u}{<} \neg x \overset{u}{<} \neg a$），我们有

$$\text{minAND}(a, b, c, d) = \neg\text{maxOR}(\neg b, \neg a, \neg d, \neg c), \text{ 及}$$
$$\text{maxAND}(a, b, c, d) = \neg\text{minOR}(\neg b, \neg a, \neg d, \neg c)$$

作为直接计算方法，代码与传播或的边界的代码非常相似，如图4-5和图4-6所示。

```
unsigned maxOR(unsigned a, unsigned b,
               unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (b & d & m) {
         temp = (b - m) | (m - 1);
         if (temp >= a) {b = temp; break;}
         temp = (d - m) | (m - 1);
         if (temp >= c) {d = temp; break;}
      }
      m = m >> 1;
   }
   return b | d;
}
```

60

图4-4 相对于x和y的边界的$x \mid y$的最大值

```
unsigned minAND(unsigned a, unsigned b,
                unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (~a & ~c & m) {
         temp = (a | m) & -m;
         if (temp <= b) {a = temp; break;}
         temp = (c | m) & -m;
         if (temp <= d) {c = temp; break;}
      }
      m = m >> 1;
   }
   return a & c;
}
```

图4-5 相对于x和y的边界的$x \& y$的最小值

使用与、或和非的边界传播方法，可以求除异或和等价之外的所有二进制逻辑表达式的边界。异或和等价的边界之所以难求，是因为当用与、或和非来表示它们时，表达式一定有两个含x和y的项。例如，假设我们要找

$$\min_{\substack{a \le x \le b \\ c \le y \le d}} (x \oplus y) = \min_{\substack{a \le x \le b \\ c \le y \le d}} ((x \,\&\, \neg y) \mid (\neg x \,\&\, y))$$

61

的边界。表达式中的或的两个操作数不能分别最小化（这里我们不给出证明，但这确实是成立的），因为我们需要寻找最小化整个或表达式的x和y的值。

下面的表达式可以用来传播异或的边界：

$$\text{minXOR}(a, b, c, d) = \text{minAND}(a, b, \neg d, \neg c) \mid \text{minAND}(\neg b, \neg a, c, d)$$

$$\text{maxXOR}(a, b, c, d) = \text{maxOR}(\mathbf{0}, \text{maxAND}(a, b, \neg d, \neg c), \mathbf{0}, \text{maxAND}(\neg b, \neg a, c, d))$$

```
unsigned maxAND(unsigned a, unsigned b,
                unsigned c, unsigned d) {
   unsigned m, temp;

   m = 0x80000000;
   while (m != 0) {
      if (b & ~d & m) {
         temp = (b & ~m) | (m - 1);
         if (temp >= a) {b = temp; break;}
      }
      else if (~b & d & m) {
         temp = (d & ~m) | (m - 1);
         if (temp >= c) {d = temp; break;}
      }
      m = m >> 1;
   }
   return b & d;
}
```

图4-6 相对于x和y的边界的$x\&y$的最大值

可以通过直接计算，算出minXOR 和maxXOR函数。除了去掉两个break语句且返回值改成a^c外，minXOR的代码与minOR（见图4-3）的代码一样。除了将if子句部分用下面的代码替换，并且返回值改成b^d之外，maxXOR的代码与maxOR（见图4-4）的代码相同。

```
temp = (b - m) | (m - 1);
if (temp >= a) b = temp;
else {
   temp = (d - m) | (m - 1);
   if (temp >= c) d = temp;
}
```

带符号边界

如果边界是带符号整数，逻辑表达式边界的传播相当复杂。如果0在a和b或在c和d之间，那么计算就是不规则的。表4-1给出了计算表达式$x \mid y$上界和下界的方法。含有“+”的项表示这一项所在栏顶端的变量的边界大于或等于0，而“-”表示该项所在栏顶端的变量的边界小于0。标有“minOR”的栏包含计算$x \mid y$的下界的表达式，最后一栏包含计算$x \mid y$的上界的表达式。对这一结果进行编码的一个方法就是根据a、b、c和d的符号位构建一个从0到15的值，并且运用“开关”语句。注意并不是0到15之间的所有值都要使用，因为不可能发生$a>b$或$c>d$的情况。

对于带符号数，下面的关系成立，

$$a \leqslant x \leqslant b \Leftrightarrow \neg b \leqslant \neg x \leqslant \neg a$$

所以，可以使用代数方法把表4-1的结果扩展到其他逻辑表达式（除异或和等价之外）的边界传播。我们将此留给读者去完成。

表4-1 通过无符号minOR和maxOR求带符号minOR和maxOR

a	*b*	*c*	*d*	**minOR**（带符号）	**maxOR**（带符号）
–	–	–	–	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)
–	–	–	+	***a***	**–1**
–	–	+	+	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)
–	+	–	–	***c***	**–1**
–	+	–	+	min(***a***, ***c***)	maxOR(**0**, ***b***, **0**, ***d***)
–	+	+	+	minOR(***a***, **0xFFFFFFFF**, ***c***, ***d***)	maxOR(**0**, ***b***, ***c***, ***d***)
+	+	–	–	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)
+	+	–	+	minOR(***a***, ***b***, ***c***, **0xFFFFFFFF**)	maxOR(***a***, ***b***, **0**, ***d***)
+	+	+	+	minOR(***a***, ***b***, ***c***, ***d***)	maxOR(***a***, ***b***, ***c***, ***d***)

63

第5章 位 计 数

5.1 1位计数

IBM Stretch计算机（ca. 1960年）有对字中1位的数目及前导0数目的计数方法。这两个量是所有逻辑操作的副产品！字中1位数目的计数函数有时被称为*种群计数指令*（例如，在Stretch及SPARCv9计算机上就如此称呼）。

对于没有这一指令的计算机，1位数目计数的好方法是，首先设置每个2位字段为原来的两个单个位的和，然后，求相邻2位字段的和，把结果放入相应的4位字段，以此类推。[RND]对这一技巧进行了更完整的讨论。图5-1给出了这一方法的示例说明，示例的第一行给出了一个要对其1位求和的计算机字，最后一行给出求和结果（十进制下的23）。

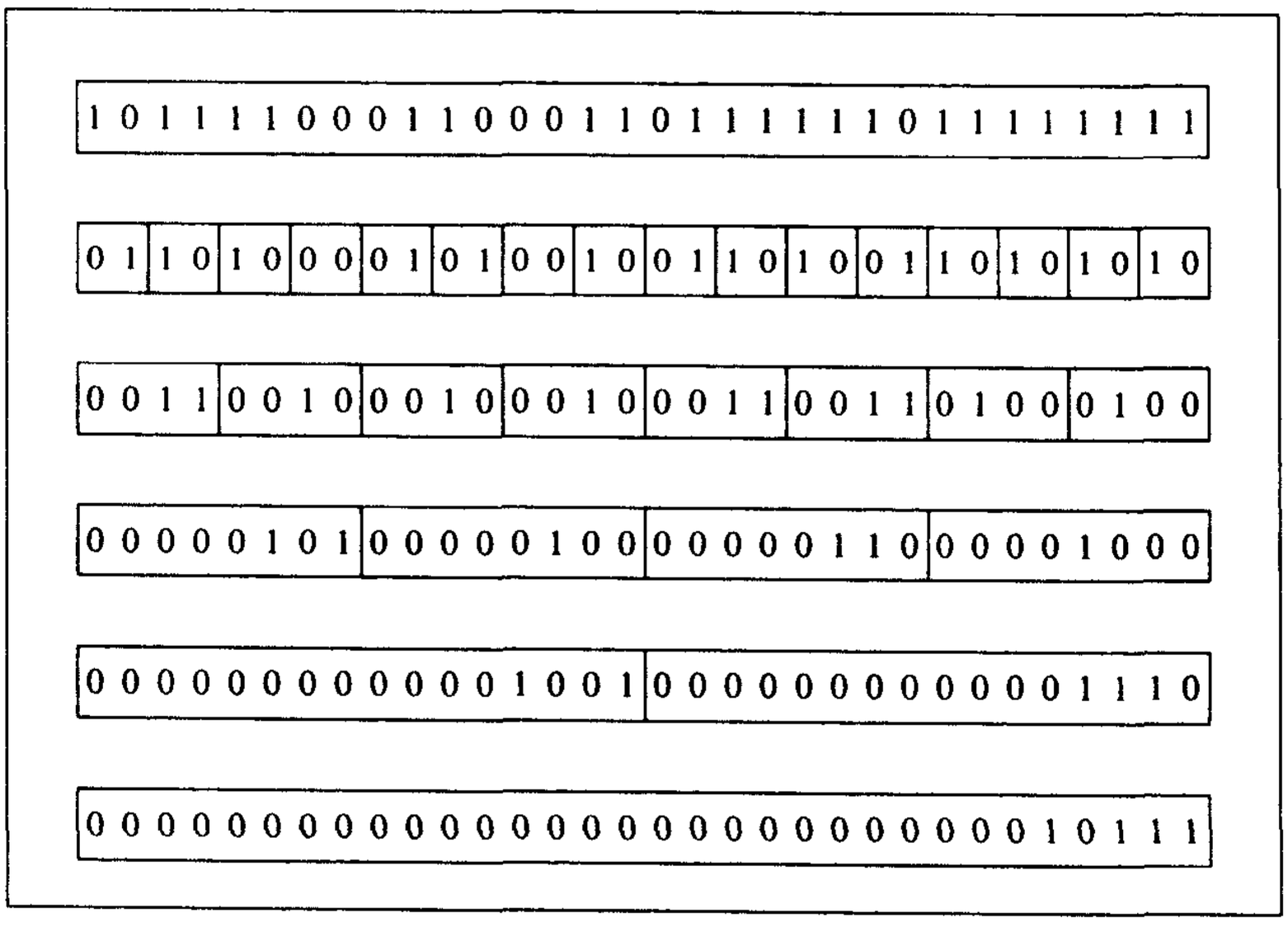


图5-1 利用“分治”策略对1位数目计数

这是一个“分治”策略的例子，在此，原始问题（对32个位求和）被分成两个问题（对16个位求和），然后分别解决这两个问题，最后将结果组合起来（在此例中，将两个结果加起来）。递归地运用这一策略，把16位字段分成8位字段，以此类推。

接下来在这种情况下，最终的小问题（对相邻位求和）可以并行解决，每一阶段的相邻和的组合也可以并行解决，阶段的数目是固定的。最终可以生成一个可以用$\log_2(32)=5$步完成的算法。

分治法的例子还有众所周知的二分查找、称为快速排序的排序方法以及7.1节将要讨论的逆置字中的位的方法，等等。

图5-1所描述的方法可以用下面C代码来实现:

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
```

代码的第一行使用了(x>>1)&0x55555555而不是或许更自然的(x&0xAAAAAAAA) >>1，这是为了避免在一个寄存器中生成两个大常量。如果计算机没有与非指令，这将导致多使用1

65 个指令。对于其他行的解释也类似。

显然，最后的与是没有必要的，而且当一个字段的和不会产生到邻近字段的进位时，相关的与也可以省略。另外，有一种可以使第一行的代码少用一个指令的方法。这导致图5-2所示的简化代码，它的执行需要21个指令，而且没有分支。

对x的第一个赋值基于下面这个相当巧妙的公式的前两项:

$$\mathrm{pop}(x) = x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \ldots - \left\lfloor \frac{x}{2^{31}} \right\rfloor \tag{5-1}$$

```
int pop(unsigned x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

66 图5-2 统计字中的1位数目

等式(5-1)中的x必须是非负的。通过把x当作无符号整数，等式(5-1)可以用31个1位立即右移位和31个减指令来实现。图5-2并行地在每一个2位字段上运用了这一等式的前两项。

下面给出等式(5-1)的简单证明。先看4位字的例子。设字是$b_3b_2b_1b_0$，其中每一个b_i=0或1。那么，

$$\begin{aligned} x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \left\lfloor \frac{x}{8} \right\rfloor &= b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ &\quad -(b_3 \cdot 2^2 + b_2 \cdot 2^1 + b_1 \cdot 2^0) \\ &\quad -(b_3 \cdot 2^1 + b_2 \cdot 2^0) \\ &\quad -(b_3 \cdot 2^0) \\ &= b_3(2^3 - 2^2 - 2^1 - 2^0) + b_2(2^2 - 2^1 - 2^0) + b_1(2^1 - 2^0) + b_0(2^0) \\ &= b_3 + b_2 + b_1 + b_0 \end{aligned}$$

或者，对于一般的32位字，因为，非负整数x的二进制表示的第i位可以由下式给出:

$$b_i = \left\lfloor \frac{x}{2^i} \right\rfloor - 2\left\lfloor \frac{x}{2^{i+1}} \right\rfloor$$

那么，通过将上述等式从i=0到31求和，可得等式(5-1)。当i=31时，上式中的后项是0，因为

$x<2^{32}$。

等式(5-1)可以推广到以其他数为底的情况，例如，以10为底，就是：

$$\text{sum\_digits}(x) = x - 9\left\lfloor \frac{x}{10} \right\rfloor - 9\left\lfloor \frac{x}{100} \right\rfloor - \ldots$$

在此，计算出所有这些项，直到它们等于0。使用上面所用的方法也可以证明以10为底的情况。

上面算法的一个变形是用以4为底而得到的与等式(5-1)类似的等式来替代图5-2中的第二个可执行行，这个替代的等式是：

```
x = x - 3*((x >> 2) & 0x33333333)
```

然而，这一代码使用与被替代行相同数目的指令，而且还需要一个快速的乘3指令。

在HAKMEN备忘录[HAK，item169]中，有一个统计字中1位数目的算法，它利用等式(5-1)的前三项生成一个3位字段的字，这个字中的每一字段存放原先该字段中1位的数目。然后，它把相邻的3位字段相加，形成一个6位字段和，再用模63运算计算字的值来把各个6位字段相加，从而得到这个字的1位数目。用C语言表示，这个算法（长常量是八进制表示）如下所示： 67

```
int pop(unsigned x) {
   unsigned n;

   n = (x >> 1) & 033333333333;          // Count bits in
   x = x - n;                            // each 3-bit
   n = (n >> 1) & 033333333333;          // field.
   x = x - n;
   x = (x + (x >> 3)) & 030707070707; // 6-bit sums.
   x = modu(x, 63);                      // Add 6-bit sums.
   return x;
}
```

最后一行使用了无符号模函数。（如果字长是3的倍数，它既可以是无符号的，也可以是带符号的。）当把字x看成以64为底的整数时，该模函数求的是6位字段的和这一事实就一目了然了。对于$b>3$，以b为底的整数除以$b-1$的余数与这些数字的和模b同余，它显然小于b。因为在这一情况下，数字的和一定小于或等于32，因此mod(x,63)一定等于x的数字和，也就是等于原来x中的1位的数目。

这一算法在DEC PDP-10计算机上只需要10个指令，因为这类计算机有这样的指令，这一指令对它的第二个操作数求余数，并且直接访问内存里的全字。在一台基本RISC计算机上，大约需要13个指令，而且要假设这类计算机有无符号模指令（但是不能直接引用立即全字或内存操作数）。但是它可能不是非常快，因为除法总是比较慢的一种操作。另外，不能通过简单地扩展常数来把它运用于64位字长，尽管它对字长直到62位的字都是可用的。

这一HAKMEM算法有一个变形，它利用等式(5-1)来计算每一个4位字段上的1位的数目，可以并行地作用于所有八个4位字段[Hay1]。于是，用一种直接的方法就可以把4位和变成8位和，然后用乘以0x0101 0101把四个字节相加。这样，我们得到：

```
int pop(unsigned x) {
   unsigned n;
```

```
    n = (x >> 1) & 0x77777777;          // Count bits in
    x = x - n;                          // each 4-bit
    n = (n >> 1) & 0x77777777;          // field.
68  x = x - n;
    n = (n >> 1) & 0x77777777;
    x = x - n;
    x = (x + (x >> 4)) & 0x0F0F0F0F;    // Get byte sums.
    x = x*0x01010101;                   // Add the bytes.
    return x >> 24;
}
```

这一代码在基本RISC计算机上需要19个指令。如果计算机是二元地址的，那么它也同样可行，因为前六行只需用一个传送寄存器指令就可实现。同样，掩码0x77777777的反复使用允许我们把它放入一个寄存器里，并利用“寄存器到寄存器”指令来引用它。而且，大部分移位的移位量都是1。

图5-3给出了一个完全不同的1位计数方法[Weg，RND]。它不断地把最右侧的1位改成0，直到结果是0。它使用2+5pop(x)个指令。如果1位的数目比较小，那么这个方法非常快。

当我们预期1位的数目非常大时，可以使用与其对称的算法。这个对称算法反复使用x=x|(x+1)把最右侧的0位改为1，直到所有位都是1（该数变为-1）。然后，它返回32-n。（或者，先对原始的x求补，或将n初始化为32再倒着计数。）

另外一个令人惊奇的算法是把x循环左移动一个位置31次，求32个项的和[MM]。这个和就是pop(x)的负值！即，

$$\mathrm{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i) \tag{5-2}$$

这里的加运算是模字长加，最后的和被解释为2的补码整数。这个算法只是新奇而已，在大多数计算机上，它都没有什么价值，因为循环要被执行31次，这样它需要63个指令再加上循环控制的开销。

```
int pop(unsigned x) {
   int n;

   n = 0;
   while (x != 0) {
      n = n + 1;
      x = x & (x - 1);
   }
   return n;
}
```

69 图5-3 稀疏种群字的1位计数

为了弄清楚等式(5-2)的工作原理，考虑它对x的一个1位所产生的作用。它循环到所有位置，对32个数字求和之后，我们得到所有位全为1的字。但是，这个字是-1。为了说明，考虑6位字长，并且设x=**001001**（二进制）：

$$\begin{array}{ll} 001001 & x \\ 010010 & x \overset{rot}{\ll} 1 \\ 100100 & x \overset{rot}{\ll} 2 \\ 001001 & x \overset{rot}{\ll} 3 \\ 010010 & x \overset{rot}{\ll} 4 \\ 100100 & x \overset{rot}{\ll} 5 \end{array}$$

当然，也可以使用*右循环移位*。

等式(5-1)的方法与上述的“循环并求和”的方法非常相似，对等式(5-1)做如下重写后，情况就更明朗了：

$$\text{pop}(x) = x - \sum_{i=1}^{31} (x \overset{u}{\gg} i)$$

这比等式(5-2)所给出的算法更好一点。它之所以更好的理由是，它使用了*右移位*，与*循环移位*指令相比，更多的计算机上拥有*右移位*指令，而且我们可以在移位量变成0时立即终止循环。这样可以减少循环控制代码，而且还可节省一些迭代。图5-4中给出了这两种算法的对比。

```
int pop(unsigned x) {
   int i, sum;

// Rotate and sum method          // Shift right & subtract

   sum = x;                       // sum = x;
   for (i = 1; i <= 31; i++) {    // while (x != 0) {
      x = rotatel(x, 1);          //    x = x >> 1;
      sum = sum + x;              //    sum = sum - x;
   }                              // }
   return -sum;                   // return sum;
}
```

图5-4 两个类似的位计数算法

70

一个无趣但可与本节pop(*x*)的所有算法一比高下的算法是拥有一个pop(*x*)值表，例如从0到255的*x*的所有pop(*x*)值。算法对这个表访问4次，对得到的4个数字求和。这个算法的一个无分支版本如下所示：

```
int pop(unsigned x) {                  // Table lookup.
   static char table[256] = {
      0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
      ...
      4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8};

   return table[x          & 0xFF] +
          table[(x >>  8) & 0xFF] +
          table[(x >> 16) & 0xFF] +
          table[(x >> 24)];
```

```
}
```

[HAK]的第167条中包含一个计算9位量中1位数目的短小算法，该算法假定这个9位量被右对齐并析出在一个寄存器中。这一算法只能在带有36位或更多位的寄存器上执行。下面给出可以在具有32位寄存器的计算机上运行的这一算法的一个版本，但它只对8位量有效。

```
x = x * 0x08040201;   // Make 4 copies.
x = x >> 3;           // So next step hits proper bits.
x = x & 0x11111111;   // Every 4th bit.
x = x * 0x11111111;   // Sum the digits (each 0 or 1).
x = x >> 28;          // Position the result.
```

计算7位量1位数目的版本如下：

```
x = x * 0x02040810;   // Make 4 copies, left-adjusted.
x = x & 0x11111111;   // Every 4th bit.
x = x * 0x11111111;   // Sum the digits (each 0 or 1).
x = x >> 28;          // Position the result.
```

在这些代码中，最后两步计算可以用x模15的余数来替换。

这些算法都不是特别理想；大多数程序员也许更喜欢查表计算。然而，后一个算法有一个使用64位算术的版本，对能够快速计算乘法的64位计算机来说，它可能很有用。它的参数是15位量。（我不相信有可以处理16位量的类似算法，除非知道不是所有16位都是1。）数据类
71 型long long是GNU C的一个扩展[Stall]，意指int的两倍长度，在这里是64位。后缀ULL指定一个无符号long long常量。

```
int pop(unsigned x) {
   unsigned long long y;
   y = x * 0x0002000400080010ULL;
   y = y & 0x1111111111111111ULL;
   y = y * 0x1111111111111111ULL;
   y = y >> 60;
   return y;
}
```

1. 计算数组中的1位数目

在没有种群计数指令的情况下，计算一个全字数组中的1位数目的最简单方法是对这个数组的每一个字运用如图5-2所给出的算法，然后，简单地将结果加起来。

另一个也许更快的方法是，将这个数组分成三个字一组的形式，对每组中的每个字执行这个算法的前两个可执行行，再把三个部分结果相加。因为在每一个4位字段上，每一个部分结果的最大值是4，把三个这样的部分结果相加，在每一个4位字段上得出一个最大值为12的字，所以不会发生到其左边字段的溢出。接下来，运用下式把这些部分结果转化成有4个8位字段的字，每个字段的最大值是24。

```
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
```

当这些字生成之后，把它们加起来直到最大值小于255；这允许我们对10个这样的字求和（⌊255/24⌋），当把10个这样的字加起来后，可以运用下面的式子将结果转变成两个16位字段的字，每一个字段的最大值是240。

```
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
```

最后，运用下面的式子把273个这样的字相加（⌊65535/240⌋），直到有必要把这个和转变成由一个32位字段组成的字为止。

```
x = (x & 0x0000FFFF) + (x >> 16);
```

在实践中，加入循环的控制指令严重降低了算法所节省的效率，因此它在很大程度上限制了我们对算法进行上述的扩展。基于这一思路，图5-5只运用了一个中间层次。首先，它生成一个包含4个8位部分和的字。然后，尽可能把这些字加起来后，生成一个全字和。相加不会产生溢出的8位字段的字的数目是⌊255/8⌋=31。 72

```
int pop_array(unsigned A[], int n) {

   int i, j, lim;
   unsigned s, s8, x;

   s = 0;
   for (i = 0; i < n; i = i + 31) {
      lim = min(n, i + 31);
      s8 = 0;
      for (j = i; j < lim; j++) {
         x = A[j];
         x = x - ((x >> 1) & 0x55555555);
         x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
         x = (x + (x >> 4)) & 0x0F0F0F0F;
         s8 = s8 + x;
      }
      x = (s8 & 0x00FF00FF) + ((s8 >> 8) & 0x00FF00FF);
      x = (x & 0x0000ffff) + (x >> 16);
      s = s + x;
   }
   return s;
}
```

图5-5 数组的1位计数

有人将这一算法与简单循环的方法进行了比较：使用GCC把两个程序编译到与基本RISC计算机相似的一台目标计算机上。结果是简单循环方法平均每个字需要22个指令，而图5-5给出的方法平均每个字需要17.6个指令，节省了20%。

2. 应用

种群计数函数的一个应用是计算两个位矢量间的“Hamming距离”，这是一个来自于错误校正代码理论的概念。Hamming距离就是矢量间取不同值的对应位的数目；即，

$$\mathrm{dist}(x, y) = \mathrm{pop}(x \oplus y)$$

参见[Dewd]中关于错误校正代码的讨论。

另外一个应用是，对于使用特定压缩方式表示的稀疏数组*A*，相当快捷地进行直接定位访问。在这一压缩表示中，只存储有定义或非零的数组元素。使用一个附加的32位字的位串数组*bits*，对于每个有定义的*A*[*i*]，*bits*的第*i*个元素是1位。作为一个加速装置，使用一个字的数组*bitsum*，*bitsum*[*j*]是前*j*个*bits*字的所有1位数目的总和。下面是当数组的第0、2、32、47、 73

48和95个元素有定义时的示例。

bits	*bitsum*	*data*
0x00000005	0	*A*[0]
0x00018001	2	*A*[2]
0x80000000	5	*A*[32]
		A[47]
		A[48]
		A[95]

给定一个下标i，$0 \leq i < 95$，数据数组的相应下标*sparse_i*由数组*bits*中相对于i的位（*bits*[*i*]）的前面的1位数目的和给出。这个和的计算如下：

```
j = i >> 5;                          // j = i/32.
k = i & 31;                          // k = rem(i, 32);
mask = 1 << k;                       // A "1" at position k.
if ((bits[j] & mask) == 0) goto no_such_element;
mask = mask - 1;                     // 1's to right of k.
sparse_i = bitsum[j] + pop(bits[j] & mask);
```

这一表示的代价是整个数组中的每一个元素有两位的开销。

种群计数函数的另一个应用是计算字中后缀0的数目（参见5.4节的“后缀0计数”）。

5.2 奇偶性

一个串的奇偶性指的是这个串包含奇数个1位还是偶数个1位。一个串如果含奇数个1位则称其具有“奇数奇偶性”，否则称其具有“偶数奇偶性”。

1. 字的奇偶性计算

这里，如果字x具有奇数奇偶性，那么我们就生成一个1，如果它具有偶数奇偶性，那么我们就生成一个0。这就是字x的各位和模2，也就是求x所有位的异或。

计算奇偶性的一个方法就是计算pop(x)；奇偶性由计算结果的最右侧位决定。如果有种群计数指令，那么这个方法很好，但是如果没有这一指令，那么存在比使用pop(x)的代码更好的方法。 74

另外一个相当直接的方法就是计算：

$$y \leftarrow \bigoplus_{i=0}^{n-1} (x \overset{u}{\gg} i)$$

其中，n是字长，字x的奇偶性由y的最右侧位决定。（表达式里的⊕表示异或，但是对这个公式，也可以使用普通的加运算。）

对于比较大的n，按如下示例的方法可以更快地计算奇偶性。（这里设n=32，移位既可以是带符号的，也可以是无符号的。）

```
y = x ^ (x >> 1);
y = y ^ (y >> 2);
y = y ^ (y >> 4);                                    (5-3)
y = y ^ (y >> 8);
y = y ^ (y >>16);
```

即使把其中蕴含的循环完全展开，这个方法也只使用10个指令。与其相比，第一个方法需要62个指令。同样，奇偶性由y的最右侧位决定。事实上，如果移位是无符号的，那么上述两种方法中的y的第i位给出了x在从第i位到其最左侧位的奇偶性。更进一步说，因为异或是其自身的逆，对于$i > j$，$y_i \oplus y_j$ ⊖是x中从第j位到第$i-1$位间的这些位的奇偶性。

这是“并行前缀”或“扫描”操作的一个例子，可以应用并行计算 [KRS; HS]。给定足够数目的处理器，它能够将表面上串行的操作步骤的时间复杂度从$O(n)$降到$O(\log_2 n)$。例如，如果有一个字数组，而且希望在这些位的整个数组上进行异或扫描操作，那么首先可以对数组的每个字使用等式(5-3)，然后，把本质上相同的技术应用于这个数组，在数组的字上做异或操作。这比简单的从左到右的操作步骤需要更多的基本异或操作，因此，对单处理器来说不是一个好方法。但是在一台带有足够多处理器的并行计算机上，它能在$O(\log_2 n)$的开销下完成这项工作，而不是在$O(n)$的开销下（这里，n是数组中字的数目）。

等式(5-3)的一个直接应用是把一个整数转换成Gray码（参见13.1节）。

如果将等式(5-3)的右移位改成左移位，那么整个字x的奇偶性最终出现在y的最左位，而且y的第i位给出x从这一位到最右侧位的奇偶性。

如果使用循环移位指令，那么如果x的奇偶性是奇数，那结果就是一个所有位为1的字；而如果x的奇偶性是偶数，则结果是所有位为0的字。 75

下面的方法用9个指令执行，x的奇偶性计算结果是整数1或0（移位是无符号的）。

```
x = x ^ (x >> 1);
x = (x ^ (x >> 2)) & 0x11111111;
x = x*0x11111111;
p = (x >> 28) & 1;
```

根据x的十六进制表示的各位的奇偶性，执行上面的第二个语句之后，x的各十六进制数字是0或1。乘指令将这些位加到一起，并把和放入高阶十六进制位的数字中。在任何十六进制列上，乘运算的加部分都不会产生进位，因为每列的最大和是8。

如果计算机有立即余指令，那么乘和移位指令可以替换为计算x除以15的余数指令，得到可用8个指令给出结果的（慢）算法。

2. 把奇偶性添加到一个7位量中

[HAK]中的第167条包含了这样一个新奇的表达式。对于右对齐并析出存放于寄存器中的7位量，这一表达式在该量上添加偶数奇偶性。具体地说，它把这个7位量的奇偶性的值放在这个7位量的七个位的左边，生成一个带有偶数奇偶性的8位量。第167条中的代码适合36位计算机，但是也可以用于32位计算机。

$$\mathrm{modu}((x * \mathbf{0x10204081}) \,\&\, \mathbf{0x888888FF}, \mathbf{1920})$$

在这里，$\mathrm{modu}(a, b)$表示a除以b的余数，参数和结果都是无符号整数，“*”表示模2^{32}乘法，常量1920是15×2^7。事实上，上面的式子计算x的各位的和，并单纯地把这个和放到x的7个位的左侧。例如，这个表达式将**0x0000 007F**映射到**0x0000 03FF**，把**0x0000 0055**映射到**0x0000 0255**。

⊖ 译者注：原书中此处是$x_i \oplus x_j$，这是错误的。因为此式与x的第$j+1$到$i-1$间的1位数目无关。而$y_i \oplus y_j$是正确的，因为$y_i = x_i \oplus x_{i+1} \oplus \cdots \oplus x_{n-1}$，$y_j = x_j \oplus x_{j+1} \oplus \cdots \oplus x_{i-1} \oplus x_i \oplus x_{i+1} \oplus \cdots \oplus x_{n-1}$，由其自逆性（以及交换性）可知$y_i \oplus y_j = x_j \oplus x_{j+1} \oplus \cdots \oplus x_{i-1}$。

IHAK中还有一个如下所示的巧妙公式，它使7位量具有奇数奇偶性：

$$\text{modu}((x * \mathbf{0x00204081}) \mid \mathbf{0x3DB6DB00}, \mathbf{1152})$$

这里，$1152=9 \times 2^7$。为了理解这个公式，需要知道8的幂模9余±1。如果把**0x3DB6 DB00**改成**0xBDB6 DB00**，那么这个公式可以用于添加偶数奇偶性。

这些方法在今天的计算机上不太适用，因为内存很便宜，而除法仍旧很慢。大多数程序员都用简单的查表法来计算这些公式。

76

3. 应用

奇偶性操作在GF(2)（在这种应用中，加操作是异或）的位矩阵乘法中非常有用。

5.3 前导0计数

有几个利用二分查找技巧统计前导0的简单方法。下面这个模式有若干版本。在基本RISC计算机上执行下面的代码需要20到29个指令。比较是“逻辑”比较（无符号整数）。

```
if (x == 0) return(32);
n = 0;
if (x <= 0x0000FFFF) {n = n +16; x = x <<16;}
if (x <= 0x00FFFFFF) {n = n + 8; x = x << 8;}
if (x <= 0x0FFFFFFF) {n = n + 4; x = x << 4;}
if (x <= 0x3FFFFFFF) {n = n + 2; x = x << 2;}
if (x <= 0x7FFFFFFF) {n = n + 1;}
return n;
```

其中一个版本是用与取代比较：

```
if ((x & 0xFFFF0000) == 0) {n = n +16; x = x <<16;}
if ((x & 0xFF000000) == 0) {n = n + 8; x = x << 8}
...
```

另外一个版本使用右移位指令来回避大的立即值。

最后的if语句就是当x的最高阶位是0时简单地在n上加1，所以可节省一个分支指令的替代方法是：

```
n = n + 1 - (x >> 31);
```

如果n被初始化为1而不是0，那么可以省略这个赋值中的“+1”。这些观察导出图5-6所示的算法（它在基本RISC计算机上需要12到20个指令）。对于x是从1位开始的情况，可以对此做进一步改进：把第一行变成

```
if ((int)x <= 0) return (~x >> 26) & 32;
```

图5-7展示了上面方法的一种反转。前导0越多它需要的操作越少，而且回避了较大的立即值和较大的移位量。在基本RISC计算机上这一方法需要12到20个指令。

77

这个算法可以结合“表辅助方式”：最后四个执行行可以用下面的代码替换。

```
static char table[256] = {0,1,2,2,3,3,3,3,4,4,...,8);
return n - table[x];
```

很多算法都得益于查表法，这里对此不再加以叙述。

为了紧凑起见，本节中的算法可以利用循环来编码。例如，可以把图5-7的算法改成图5-8

所示的算法。图5-8的算法需要23到33个基本RISC指令，其中10个是条件分支指令。

```
int nlz(unsigned x) {
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x >> 16) == 0) {n = n +16; x = x <<16;}
   if ((x >> 24) == 0) {n = n + 8; x = x << 8;}
   if ((x >> 28) == 0) {n = n + 4; x = x << 4;}
   if ((x >> 30) == 0) {n = n + 2; x = x << 2;}
   n = n - (x >> 31);
   return n;
}
```

图5-6 前导0数目，二分查找

```
int nlz(unsigned x) {
   unsigned y;
   int n;

   n = 32;
   y = x >>16;  if (y != 0) {n = n -16;  x = y;}
   y = x >> 8;  if (y != 0) {n = n - 8;  x = y;}
   y = x >> 4;  if (y != 0) {n = n - 4;  x = y;}
   y = x >> 2;  if (y != 0) {n = n - 2;  x = y;}
   y = x >> 1;  if (y != 0) return n - 2;
   return n - x;
}
```

图5-7 前导0数目，二分查找，向下计数

```
int nlz(unsigned x) {
   unsigned y;
   int n, c;

   n = 32;
   c = 16;
   do {
      y = x >> c;  if (y != 0) {n = n - c;  x = y;}
      c = c >> 1;
   } while (c != 0);
   return n - x;
}
```

图5-8 前导0数目，二分查找，用循环编码

当然，可以简单地一次左移位一个位置并计数，直到符号位变为1；也可以一次右移位一个位置，直到字的各位都是0。如果我们分别预期前导0的数目很小或很大，那么这些算法很紧凑，运行得也很好。可以把这些方法结合起来，如图5-9所示。我们提及这个算法是因为结 78
合两个算法并选择首先停止的结果这一技术更具应用性。由于相互独立的指令相邻，这导致

代码在超标量计算机和VLIW计算机上运行得更快。(这些计算机能同时执行两个或更多个相互独立的指令。)

```
int nlz(int x) {
   int y, n;

   n = 0;
   y = x;
L: if (x < 0) return n;
   if (y == 0) return 32 - n;
   n = n + 1;
   x = x << 1;
   y = y >> 1;
   goto L;
}
```

图5-9 前导0数目，从两端同时计算

在基本RISC计算机上，图5-9的算法在$\min(3+6\text{nlz}(x), 5+6(32-\text{nlz}(x)))$个指令内执行，或在最坏的情况下在99个指令内执行。然而，我们可以想象，如果比较结果是移位的副产品，那么超标量计算机或VLIW计算机可以在1个周期内执行整个循环体，加上分支的额外开销则在2个周期内执行整个循环体。

把图5-7或图5-8的算法改成无分支版本非常简单。图5-10给出了在28个基本RISC指令内执行的无分支版本。

```
int nlz(unsigned x) {
   int y, m, n;

   y = -(x >> 16);          // If left half of x is 0,
   m = (y >> 16) & 16;      // set n = 16. If left half
   n = 16 - m;              // is nonzero, set n = 0 and
   x = x >> m;              // shift x right 16.
                            // Now x is of the form 0000xxxx.
   y = x - 0x100;           // If positions 8-15 are 0,
   m = (y >> 16) & 8;       // add 8 to n and shift x left 8.
   n = n + m;
   x = x << m;

   y = x - 0x1000;          // If positions 12-15 are 0,
   m = (y >> 16) & 4;       // add 4 to n and shift x left 4.
   n = n + m;
   x = x << m;

   y = x - 0x4000;          // If positions 14-15 are 0,
   m = (y >> 16) & 2;       // add 2 to n and shift x left 2.
   n = n + m;
   x = x << m;

   y = x >> 14;             // Set y = 0, 1, 2, or 3.
   m = y & ~(y >> 1);       // Set m = 0, 1, 2, or 2 resp.
   return n + 2 - m;
}
```

图5-10 前导0数目，无分支二分查找

如果计算机上有种群计数指令，那么图5-11给出了一个计算前导0数目的函数的好方法。可以逆置x的5个赋值，也可以按任意顺序去赋值。图5-11所给出的算法是无分支的，需要11个指令。即使种群计数不可用，这个算法可能也很有用。使用图5-2给出的1位计数的21指令代码，图5-11的算法需要32个无分支基本RISC指令。 79

```
int nlz(unsigned x) {
   int pop(unsigned x);

   x = x | (x >> 1);
   x = x | (x >> 2);
   x = x | (x >> 4);
   x = x | (x >> 8);
   x = x | (x >>16);
   return pop(~x);
}
```

图5-11 前导0数目，右传播和1位计数 80

1. 浮点方法

浮点的后正规化功能（post-normalization facility）可以用于前导0计数。它可以很好地应用于IEEE格式的浮点数。思路是，把给定的无符号整数转换成双精度浮点数，提取它的指数，从一个常量中减去它。图5-12给出了这样的一个完整的程序。

```
int nlz(unsigned k) {
   union {
      unsigned asInt[2];
      double asDouble;
   };
   int n;

   asDouble = (double)k + 0.5;
   n = 1054 - (asInt[LE] >> 20);
   return n;
}
```

图5-12 前导0数目，使用IEEE浮点数

这个代码使用C++的“匿名union”类型来用双精度浮点量覆盖整数。在一个小端方式的计算机上执行时，变量LE必须是1，而在大端方式的计算机上，变量LE必须是0。在k上加上0.5或其他更小的数是为了使算法在k=0时也能正确计算。

我们不去评估这个算法的运行时间，因为不同计算机的浮点性质也不同。例如，很多计算机有它们自己的不同于整数寄存器的浮点寄存器，在这样的计算机上，通过内存的数据传送可能需要将整数转换成浮点数，并将结果传送到整数寄存器。

根据ANSI标准，图5-12所示的代码不是正确的C或C++代码，因为代码中两个不同的类型访问同一个内存；因此，无法保证它能在特定计算机和特定编译器上运行。无论使用什么优化标准，在AIX上使用IBM的XLC编译器以及在AIX和Windows 2000上使用GCC编译器，它都能正常工作。如果把代码改成使用下面的代码去定义覆盖:

```
xx = (double)k + 0.5;
n = 1054 - (*((unsigned *)&xx + LE) >> 20);
```

那么当开启优化功能时，它不能编译。这个代码违反了二级ANSI标准，也就是说，只能对指向数组元素的指针做指针算术[Cohen]。然而，编译失败的原因在于它违反了涉及重载定义的
81 一级ANSI标准。

尽管这个代码脆弱[⊖]，但是下面给出它的三个变形。

```
asDouble = (double)k;
n = 1054 - (asInt[LE] >> 20);
n = (n & 31) + (n >> 9);

k = k & ~(k >> 1);
asFloat = (float)k + 0.5f;
n = 158 - (asInt >> 23);

k = k & ~(k >> 1);
asFloat = (float)k;
n = 158 - (asInt >> 23);
n = (n & 31) + (n >> 6);
```

在第一个变形中，我们不是通过浮点加0.5来解决k=0的问题的，而是通过在结果n上做整数算术（如果不进行修正，那么n可以是1054或0x41E）。

后两个变形使用了单精度浮点数，“匿名union”也做相应改变。这里，产生了一个新问题：如果舍入模式是舍入到最近（通常都是这样的）或向+∞舍入，那么舍入可能扔掉结果。对于舍入到最近的舍入模式，对以下范围的k会出现问题：从FFFF FF80到FFFF FFFF、从7FFF FFC0到7FFF FFFF、从3FFF FFE0到3FFF FFFF，等等。在舍入过程中，加1会一直向左产生进位，改变最高有效1位的位置。上述所用的修正步骤把最高有效1位右边的位清零，防止进位。

GNU C/C++编译器有一个独一无二的特性，它允许将任何这样的方案作为宏指令进行编码、对函数引用提供内嵌代码[Stall]。这一特性允许我们将语句，包括声明语句，插入到调用表达式的代码中。这样的语句序列通常以一个充当这一语句序列的值的表达式结束。下面是对应于单精度变形的宏定义。（在C语言中，习惯用大写字母来命名宏）。

```
#define NLZ(k) \
   ({union {unsigned _asInt; float _asFloat;}; \
     unsigned _kk = (k) & ~((unsigned)(k) >> 1); \
     _asFloat = (float)_kk + 0.5f; \
     158 - (_asInt >> 23);})
```

82 我们使用下划线来避免与参数k的名字混淆；可以假定，用户定义的名字不以下划线开始。

2. 与对数函数的关系

“nlz”函数本质上就是“以2为底的整数对数”函数。对于无符号整数$x \neq 0$，有：

$$\lfloor \log_2(x) \rfloor = 31 - \mathrm{nlz}(x), \quad \text{及}$$
$$\lceil \log_2(x) \rceil = 32 - \mathrm{nlz}(x - 1)$$

⊖ 它的脆弱是由C的用法引起的。如果用特定机器的机器语言编码，或使用编译器生成特定机器的机器代码，那么这一方法完全可用。

参见11.4节。

另一个密切相关的函数是bitsize，这一函数计算将其参数表示为2的补码形式的带符号量时所需的位的数目。它的定义如下：

$$\text{bitsize}(x) = \begin{cases} 1, & x = -1 \text{ 或 } 0, \\ 2, & x = -2 \text{ 或 } 1, \\ 3, & -4 \leqslant x \leqslant -3 \text{ 或 } 2 \leqslant x \leqslant 3, \\ 4, & -8 \leqslant x \leqslant -5 \text{ 或 } 4 \leqslant x \leqslant 7, \\ \cdots & \cdots \\ 32, & -2^{31} \leqslant x \leqslant -2^{30}+1 \text{ 或 } 2^{30} \leqslant x \leqslant 2^{31}-1 \end{cases}$$

根据定义，有bitsize(x)=bitsize($-x-1$)。但是，$-x-1=\neg x$，所以bitsize的一个算法是（移位是带符号移位）：

```
x = x ^ (x >> 31);          // If (x < 0) x = -x - 1;
return 33 - nlz(x);
```

3. 应用

前导0数目函数的两个重要应用是模拟浮点算术操作和各种除运算算法（参见图9-1和图9-3）。这一指令似乎还有一些其他的应用。

利用前导0数目函数，我们可以得到仅需3个指令的“$x=y$”谓词（参见2.11节的“比较谓词”）。它在计算某些初等函数时也有用（参见第11章）。

一个新颖的应用是，通过生成均匀分布的随机整数并对结果取“nlz”来生成按指数分布的随机整数[GLS1]。结果为0的概率是1/2，结果为1的概率是1/4，结果为2的概率是1/8，依此类推。另外一个应用是，辅助在字中查找特定长度的连续1位串或连续0位串，这一查找可以用于某些磁盘块分配算法。对于最后两个应用，也可以使用后缀0数目函数。 83

5.4 后缀0计数

如果前导0数目函数可用，那么后缀0计数的最好方法可能就是把它转化成前导0计数问题：

$$32 - \text{nlz}(\neg x \,\&\, (x-1))$$

如果种群计数指令可用，更好的方法就是构造识别后缀0的掩码，然后对其中的1位计数[Hay2]，例如，

$$\text{pop}(\neg x \,\&\, (x-1)), \text{ 及}$$

$$32 - \text{pop}(x \mid -x)$$

还有构造识别x的后缀0的掩码的其他表达式，例如2.1节中给出的那些表达式。即使计算机没有任何位计数指令，这些方法也是可接受的。利用图5-2给出的pop(x)的算法，上面的第一个表达式大约需要3+21=24个指令（无分支）。

图5-13给出的算法直接完成后缀0计数，（对$x \neq 0$）大约需要12到20个基本RISC指令。

如果编译器不能聪明地将n+16简化成17，你可以直接将n+16简化成17。（鉴于我们对指令的计数方式，这对指令的数目不产生影响）。

```
int ntz(unsigned x) {
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x & 0x0000FFFF) == 0) {n = n +16; x = x >>16;}
   if ((x & 0x000000FF) == 0) {n = n + 8; x = x >> 8;}
   if ((x & 0x0000000F) == 0) {n = n + 4; x = x >> 4;}
   if ((x & 0x00000003) == 0) {n = n + 2; x = x >> 2;}
   return n - (x & 1);
}
```

图5-13 后缀0的数目，二分查找

图5-14给出了使用较小立即值和简单操作的一个变形。它需要12到21个基本RISC指令。同上面的算法不同，当后缀0的数目很小时，图5-14的算法执行较多指令，但是分支“落空(fall through)”的数目也更多。

```
int ntz(unsigned x) {
   unsigned y;
   int n;

   if (x == 0) return 32;
   n = 31;
   y = x <<16;  if (y != 0) {n = n -16;  x = y;}
   y = x << 8;  if (y != 0) {n = n - 8;  x = y;}
   y = x << 4;  if (y != 0) {n = n - 4;  x = y;}
   y = x << 2;  if (y != 0) {n = n - 2;  x = y;}
   y = x << 1;  if (y != 0) {n = n - 1;}
   return n;
}
```

图5-14 后缀0的数目，较小的立即值

上面的return语句也可以编码成:

```
n = n - ((x << 1) >> 31);
```

它可以节省一个分支，但是不能节省指令数目。

从执行的指令数目来看，“查找树”更占优势 [Aus2]。图5-15对8位参数展示了这一算法。除了最后两个（return 7或return 8）路径外，这一算法对其他所有路径都需要7个指令，对最后两个路径它需要9个指令。32位版本需要11到13个指令。不幸的是，当字长较大时，这个程序也相当大。上面的8位版本有12行可执行源代码并被编译成41个指令的目标程序。32位版本的源代码有48行可执行源代码并被编译成164个指令的目标程序。64位版本的数据是32位版本的两倍。

如果能够预期后缀0的数目很小或很大，那么如下所示的简单循环很快。左侧的算法大约需要5+3ntz(x)个基本RISC指令，而右侧则需要3+3(32−ntz(x))个基本RISC指令。

84

如果数x是均匀分布的，那么后缀0的平均数目非常接近1.0，这一点非常有趣。为了弄明白这一点，设p_i是正好有n_i个后缀0的概率，乘积$p_i n_i$的和为:

```
int ntz(char x) {
   if (x & 15) {
      if (x & 3) {
         if (x & 1) return 0;
         else return 1;
      }
      else if (x & 4) return 2;
      else return 3;
   }
   else if (x & 0x30) {
      if (x & 0x10) return 4;
      else return 5;
   }
   else if (x & 0x40) return 6;
   else if (x) return 7;
   else return 8;
}
```

图5-15 后缀0的数目，二分查找树

```
int ntz(unsigned x) {
   int n;

   x = ~x & (x - 1);
   n = 0;                          // n = 32;
   while(x != 0) {                 // while (x != 0) {
      n = n + 1;                   //    n = n - 1;
      x = x >> 1;                  //    x = x + x;
   }                               // }
   return n;                       // return n;
}
```

图5-16 后缀0的数目，简单计数循环

$$S \equiv \frac{1}{2}\times 0 + \frac{1}{4}\times 1 + \frac{1}{8}\times 2 + \frac{1}{16}\times 3 + \frac{1}{32}\times 4 + \frac{1}{64}\times 5 + \ldots$$

$$\equiv \sum_{n=0}^{\infty} \frac{n}{2^{n+1}}$$

为了计算这个和，考虑下面的数组：

$$\begin{array}{ccccccc} 1/4 & 1/8 & 1/16 & 1/32 & 1/64 & \ldots \\ & 1/8 & 1/16 & 1/32 & 1/64 & \ldots \\ & & 1/16 & 1/32 & 1/64 & \ldots \\ & & & 1/32 & 1/64 & \ldots \\ & & & & 1/64 & \ldots \\ & & & & & \ldots \end{array}$$

数组中每一列的和都是级数S的一个项。因此S是这个数组中所有项的和。各行的和如下：

$$1/4 + 1/8 + 1/16 + 1/32 + \ldots = 1/2$$
$$1/8 + 1/16 + 1/32 + 1/64 + \ldots = 1/4$$
$$1/16 + 1/32 + 1/64 + 1/128 + \ldots = 1/8$$
$$\ldots$$

85
/
86

而这些结果的和是1/2+1/4+1/8+...=1。原级数的绝对收敛性保证重新排列是正确的。

有时，需要一个与ntz(x)相似的函数，只是参数为0时不同，其结果也许是个错误，需要能够从这一函数的“正常”值中容易区分出来。例如，让我们定义计算“x中2的因子的数目”的函数为：

$$\text{nfact2}(x) = \begin{cases} \text{ntz}(x), & x \neq 0 \\ -1, & x = 0 \end{cases}$$

这一函数可以通过下式计算

$$31 - \text{nlz}(x \;\&\; -x)$$

应用

[GLS1]指出了后缀0数目函数的一些有趣的应用。Eric Jensen把它命名为“标尺函数”，因为通过它给出了标尺上二等分、四等分、八等分等等的刻度线高度。

它在R. W. Gosper的循环检测算法中有应用。以下对循环检测算法做较详细的说明，因为它相当精致，而且比初看起来更有用。

假设序列$X_0, X_1, X_2, \ldots$是由$X_{n+1}=f(X_n)$定义的。如果f的值域是有限的，那么序列必定是循环的。也就是说，这个序列包含一个前导序列$X_0, X_1, \ldots, X_{\mu-1}$，后面跟着一个不受限制的循环$X_\mu, X_{\mu+1}, \ldots, X_{\mu+\lambda-1}$（$X_\mu=X_{\mu+\lambda}$，$X_{\mu+1}=X_{\mu+\lambda+1}$，依此类推，其中λ是循环的周期）。给定函数$f$，循环检测问题就是要找到第一个重复出现的元素的下标μ和周期λ。在测试随机数的生成以及检测链表中的循环等方面，循环检测都有应用。

我们可以在生成序列的值时把它们保存起来，并且将每个新元素与以前的元素相比较。这一比较可以立即显示第二个循环的开始位置。但是，存在不论在空间还是时间上都更有效的算法。

最简单的算法也许应该是R. W. Floyd[Knu2, 3.1节的问题6]给出的算法。这一算法重复如下过程：

$$x = f(x)$$
$$y = f(f(y))$$

其中x、y初始化为X_0。经过n步之后，$x=X_n$、$y=X_{2n}$。比较这些元素，如果相等，那么就知X_n和X_{2n}的间隔是周期λ的整数倍，即$2n-n=n$是λ的倍数。于是μ就通过重新生成序列并比较X_0和

87

X_n，X_1和X_{n+1}，等等来确定。当X_μ和$X_{n+\mu}$相比较时出现相等的情况。最后，通过重新生成更多的元素并把X_μ同$X_{\mu+1}, X_{\mu+2}, \ldots$相比较来决定λ。这一算法仅需要较小且有界的空间，但是它要多次计算f。

Gosper的算法[HAK，item132；Knu2，3.1节练习、问题7的答案]寻找周期λ，但不求第一次循环开始点μ。它的主要特征是从不回头去重新计算f，而且在空间和时间上相当经济。它所用的空间是无界的；它需要一个长度为$\log_2(\Lambda)+1$的表，其中Λ是最大可能周期。这不需要太多的空间；例如，如果已知$\Lambda < 2^{32}$，那么33个字就足够了。

Gosper算法的C语言代码如图5-17所示。这里所给出的C函数以需要分析的函数f和初始值X_0为参数。它返回μ的上界和下界以及周期λ。（尽管Gosper的算法不能计算μ，但是它能计算μ的下界μ_l和上界μ_u，$\mu_u-\mu_l+1 \leq \max(\lambda-1, 1)$）。对于$n=1,2\cdots$，算法把$X_n$同下面集合中的元素相比较，这个集合由满足下面条件的$\lfloor \log_2 n \rfloor + 1$个$X_i$组成：或者$X_i$是在$X_n$之前且使$i+1$结束于1位的最

接近X_n的元素（也就是说，i是小于n的最大偶数），或者X_i是在X_n之前且使$i+1$结束于一个0位的最接近X_n的元素，或者X_i是在X_n之前且使$i+1$结束于两个0位的最接近X_n的元素，以此类推。

```
void ld_Gosper(int (*f)(int), int X0, int *mu_l,
                               int *mu_u, int *lambda) {
   int Xn, k, m, kmax, n, lgl;
   int T[33];

   T[0] = X0;
   Xn = X0;
   for (n = 1; ; n++) {
      Xn = f(Xn);
      kmax = 31 - nlz(n);              // Floor(log2 n).
      for (k = 0; k <= kmax; k++) {
         if (Xn == T[k]) goto L;
      }
      T[ntz(n+1)] = Xn;                // No match.
   }
L:
   // Compute m = max{i | i < n and ntz(i+1) = k}.

   m = ((((n >> k) - 1) | 1) << k) - 1;
   *lambda = n - m;
   lgl = 31 - nlz(*lambda - 1); // Ceil(log2 lambda) - 1.
   *mu_u = m;                           // Upper bound on mu.
   *mu_l = m - max(1, 1 << lgl) + 1;// Lower bound on mu.
}
```

图5-17 Gosper的循环检测算法 88

因此，比较过程如下：

$X_1 : X_0$　　$X_7 : X_6, X_5, X_3$　　$X_{13} : X_{12}, X_9, X_{11}, X_7$

$X_2 : X_0, X_1$　　$X_8 : X_6, X_5, X_3, X_7$　　$X_{14} : X_{12}, X_{13}, X_{11}, X_7$

$X_3 : X_2, X_1$　　$X_9 : X_8, X_5, X_3, X_7$　　$X_{15} : X_{14}, X_{13}, X_{11}, X_7$

$X_4 : X_2, X_1, X_3$　　$X_{10} : X_8, X_9, X_3, X_7$　　$X_{16} : X_{14}, X_{13}, X_{11}, X_7, X_{15}$

$X_5 : X_4, X_1, X_3$　　$X_{11} : X_{10}, X_9, X_3, X_7$　　$X_{17} : X_{16}, X_{13}, X_{11}, X_7, X_{15}$

$X_6 : X_4, X_5, X_3$　　$X_{12} : X_{10}, X_9, X_{11}, X_7$　　$X_{18} : X_{16}, X_{17}, X_{11}, X_7, X_{15}$

可以证明，Gosper的循环检测算法总是结束于第二个循环中的某个n处，也就是说，$n<\mu+2\lambda$。细节参见[Knu2]。

标尺函数揭示出如何解汉诺塔之谜。从0到$n-1$对n个圆盘计数。对于从1到2^n-1的k，第k次移动圆盘时，将圆盘ntz(k)循环移动至其右邻的圆柱上。

标尺函数可以用来生成反射二进制Gray编码（参见13.1节）。开始于任意的n位字，对于从1到2^n-1，在第k步，反转第ntz(k)位。 89

第6章 字 搜 索

6.1 寻找第一个0字节

寻找第一个0字节的函数最初主要起因于C语言中字符串的表示方式。C语言不明确存储字符串的长度；而是在字符串的结尾用一个各位都是0的字节做标记。C程序使用“strlen”（串长度）函数来求字符串的长度。这个函数从左到右搜索串中的0字节，返回扫描到的字节数目，不计最后的0字节。

“strlen”的快速实现也许就是装入并检测单一字节，直到达到字的边界，然后是一次把一个字装入寄存器中，并检测寄存器中0字节的存在。在大端方式的计算机上，我们希望有返回左边第一个0字节的下标的函数。一个便利的编码是用0到3分别表示第0到第3个字节是0字节，用4表示在这个字中没有0字节。把串长初始化为0，并当搜索完一个字后，把这个值加到串长。在小端方式的计算机上，则需要求从寄存器右端开始的第一个0位的下标，因为当把字装入到寄存器中时，小端方式的计算机要颠倒字中的四个字节。具体地说，我们要关注下面的函数，其中“00”表示一个0字节，“nn”表示一个非0字节，“xx”表示一个或者是0或者是非0的字节。

$$\text{zbytel}(x) = \begin{cases} 0, & x = 00xxxxxx, \\ 1, & x = nn00xxxx, \\ 2, & x = nnnn00xx, \\ 3, & x = nnnnnn00, \\ 4, & x = nnnnnnnn. \end{cases} \qquad \text{zbyter}(x) = \begin{cases} 0, & x = xxxxxx00, \\ 1, & x = xxxx00nn, \\ 2, & x = xx00nnnn, \\ 3, & x = 00nnnnnn, \\ 4, & x = nnnnnnnn. \end{cases}$$

第一个求最左0字节函数的过程如图6-1所示。该过程按从左到右的顺序，简单地检测每一字节，当找到第一个0字节时，返回检测的结果。

这一过程需要2到11个基本RISC指令，需要11个指令的情况是这个字中没有0字节（对“strlen”函数来说这是一个重要的情况）。求最右0字节的算法与此非常相似。

图6-2给出了这一函数的一个无分支过程。其思路是把每个0字节转换成0x80，把每个非0字节转换成0x00，然后利用前导0数目指令。如果计算机有前导0数目和或非指令的话，这个过程需要8个指令。[Lamp]中有一些类似的技巧。


```
int zbytel(unsigned x) {
   if              ((x >> 24) == 0) return 0;
   else if ((x & 0x00FF0000) == 0) return 1;
   else if ((x & 0x0000FF00) == 0) return 2;
   else if ((x & 0x000000FF) == 0) return 3;
   else return 4;
}
```

图6-1 求最左0字节，简单的检测序列

```
int zbytel(unsigned x) {
   unsigned y;
   int n;
                              // Original byte: 00 80 other
   y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;   // 7F 7F 1xxxxxxx
   y = ~(y | x | 0x7F7F7F7F);           // 80 00 00000000
   n = nlz(y) >> 3;                     // n = 0 ... 4, 4 if x
   return n;                            // has no 0-byte.
}
```

图6-2 求最左0字节，无分支代码

最右0字节的位置可以由上述计算中y的最终值中存放的后缀0数目整除8（舍弃小数部分）来计算。通过前导0数目指令求后缀0的表达式（参见5.4节），可以用下面的式子替换上面过程中对n的赋值来计算。

```
n = (32 - nlz(~y & (y - 1))) >> 3;
```

如果计算机有或非和与非指令的话，这是求最右0字节的位置的一个12指令解。

在大多数PowerPC计算机中，不需要寻找最右0字节的过程。而是使用反向字节字装入（lwbrx）指令来装入字。

图6-2所示的过程在64位计算机上比在32位计算机上更有价值，因为在64位计算机上，过程（经过直截了当的修改）需要大约相同数目的指令（根据常量的生成方式，需要7个或10个指令），然而图6-1所示的技术（在最坏情况下）需要23个指令。

如果只希望做0字节存在的检测，那么在第二个对y的赋值之后插入零分支或非零分支语句。

当“nlz”指令不可用时，似乎没有寻找第一个0字节的好方法。图6-3给出了一个可能的

92 方法（只给出了代码的可执行部分）。

```
                              // Original byte: 00 80 other
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F; // 7F 7F 1xxxxxxx
y = ~(y | x | 0x7F7F7F7F);          // 80 00 00000000
                                    // These steps map:
if (y == 0) return 4;               // 00000000 ==> 4,
else if (y > 0x0000FFFF)            // 80xxxxxx ==> 0,
   return (y >> 31) ^ 1;            // 0080xxxx ==> 1,
else                                // 000080xx ==> 2,
   return (y >> 15) ^ 3;            // 00000080 ==> 3.
```

图6-3 寻找最左0字节，不使用nlz指令

它的执行需要10到13个基本RISC指令，所有字节都是非零字节时需要10个指令。因此，它可能不如图6-1所示的代码好，尽管它需要的分支指令较少。这种方式不太适合64位计算机。

还有其他回避“nlz”函数的可能性。由图6-3的代码所计算的y值由四个字节组成，每个字节或者是0x00，或者是0x80。通过求这样的数对0x7F的余数，把原始值中的最多4个1位移动并压缩到最右侧4个位上。因此，余数的值域是0到15，而且惟一地反映原始值。例如，

remu(**0x80808080**, **127**) = **15**

remu(**0x80000000**, **127**) = **8**

remu(**0x00008080**, **127**) = **3**，等等

这个值可以用于索引含16个字节的表，从而得出希望的结果。因此，从if(y==0)开始的代码可以用下面的代码取代。

```
static char table[16] = {4, 3, 2, 2, 1, 1, 1, 1,
                         0, 0, 0, 0, 0, 0, 0, 0};
return table[y%127];
```

其中y是无符号量。也可以用31来取代127，但是要使用不同的表。

涉及到除以127或31的这些方法实际上只是出于好奇，因为即使是在硬件上直接实现，求余函数往往也需要至少20个周期。然而，下面是对图6-3中以if(y==0)开始的代码的一个更有效的替换:

```
return table[hopu(y, 0x02040810) & 15];
return table[y*0x00204081 >> 28];
```

在此，hopu(a,b)表示a和b的无符号积的高阶32位。在第二行，我们假设按通常的HLL惯例， 93
即乘运算的值取完全积的低阶32位。无论计算机用快速的乘运算，还是通过“移位和加”来实现乘以0x204081，这都是一个实用的方法。下面的式子暗示它可以用四个指令完成:

$$y(1 + 2^7 + 2^{14} + 2^{21}) = y(1 + 2^7)(1 + 2^{14})$$

利用这样的4个周期乘运算，过程所需的总时间是13个周期（其中7个用来计算y，4个用来实现“移位和加”，另2个用来实现28位的右移位和表索引），当然它是无分支的。

这些尺度也适用于64位计算机。对“模”方法，使用

```
return table[y%511];
```

其中，表的长度是256，表中元素是8、0、1、0、2、0、1、0、3、0、1、0、2、0、1、0、4……（即，table[i]等于*i*中后缀0的数目）。

对乘积方法，使用

```
return table[hopu(y, 0x0204081020408100) & 255]; 或
return table[(y*0x0002040810204081)»56];
```

其中，表的长度是256，表中元素是8、7、6、6、5、5、5、5、4、4、4、4、4、4、4、4、3……乘以0x20408 10204081的运算可以用下面的式子来完成。

$$t_1 \leftarrow y(1 + 2^7)$$

$$t_2 \leftarrow t_1(1 + 2^{14})$$

$$t_3 \leftarrow t_2(1 + 2^{28})$$

它给出了一个13周期解。

所有使用表的变形当然可以通过简单地修改表中的数据来实现寻找最右0字节函数。

对于没有或非指令的32位计算机，可以省略图6-3中第二个对y的赋值语句中的非，而对table[i]=0，0，0，0，0，0，0，0，1，1，1，1，2，2，3，4，使用上面给出的三个return语句中的一个。这种模式不能在64位计算机上很好地工作。

图6-2的过程有一个有趣的变形，其目标计算机仍然是没有前导0数目指令的计算机。设*a*、*b*、*c*和*d*是1位变量，分别对应于谓词“*x*的第一字节是非零的”、“*x*的第二个字节是非零的”，以此类推。那么，

$$\text{zbytel}(x) = a + ab + abc + abcd$$

94 其中的乘法可以通过与指令来实现，由此得图6-4所示的过程（只给出可执行代码）。

```
y = (x & 0x7F7F7F7F) + 0x7F7F7F7F;
y = y | x;                    // Leading 1 on nonzero bytes.

t1 =  y >> 31;                        // t1 = a.
t2 = (y >> 23) & t1;                  // t2 = ab.
t3 = (y >> 15) & t2;                  // t3 = abc.
t4 = (y >>  7) & t3;                  // t4 = abcd.
return t1 + t2 + t3 + t4;
```

图6-4 通过计算多项式寻找最左0字节

这一过程需要15个基本RISC指令，它并不特别快，但有一定数量的并行。在可以并行执行三个独立指令的超标量计算机上，仅需要10个周期。

基于下面的公式，这一过程有一个计算寻找最右0字节函数的简单变形：

$$\text{zbyter}(x) = abcd + bcd + cd + d$$

（这比图6-4所示的代码多1个与指令。）

1. 一些简单的推广

函数“zbytel”和“zbyter”可以用于搜索等于任意给定值的字节，首先对变量x与一个每个字节都是给定值的字取异或。例如，为了搜索x中的ASCII空格（0x20），搜索$x \oplus$0x2020 2020中的0字节。

类似，为了搜索两个字x和y中相等字节的位置，可以搜索$x \oplus y$中的0字节。

图6-2所示的代码及其变形对字节边界没有特殊要求。例如，为了搜索一个字，看它第一个4位、随后的12位或最后16位中是否有0值，可以用0x77FF7FFF取代图6-2代码的掩码[PHO]。（如果字段的长度是1，那么让掩码在该位置为0）。

2. 搜索给定范围内的值

很容易将图6-2的代码修改成搜索范围在0和比128小的任意特定值之间的字节。为了说明，下面给出寻找取0到9的值的最左侧字节的下标的代码：

```
y = (x & 0x7F7F7F7F) + 0x76767676;
y = y | x;
y = y | 0x7F7F7F7F;           // Bytes > 9 are 0xFF.
y = ~y;                       // Bytes > 9 are 0x00,
                              // Bytes <= 9 are 0x80.
n = nlz(y) >> 3;
```

95

更一般地，假设希望寻找这样的字，它的最左侧字节的值在a到b之间，a与b之间的差小于128。例如，大写字母在ASCII编码中的范围在0x41到0x5A之间。要寻找字中的第一个大写字母，用不越过字节边界传播借位的方式来减0x4141 4141，再使用上面的代码来识别取值在0到0x19（0x5A到0x41）之间的字节。对y=0x41414141使用2.17节给出的减法公式的简化形式，可得：

```
d = (x | 0x80808080) - 0x41414141;
d = ~((x | 0x7F7F7F7F) ^ d);
y = (d & 0x7F7F7F7F) + 0x66666666;
```

```
y = y | d;
y = y | 0x7F7F7F7F;     // Bytes not from 41-5A are FF.
y = ~y;                 // Bytes not from 41-5A are 00,
                        // Bytes from 41-5A are 80.
n = nlz(y) >> 3;
```

对某些值的范围，有更简单的代码。例如，要寻找取值在0x30到0x39之间（十进制数字的ASCII编码）的第一个字节，只要简单地取输入字与0x3030 3030的异或，然后利用上面给出的代码搜索范围在0到9之间的值即可。（这一简化适用于上界和下界有n个相同的高阶位，而且下界以$8-n$个0结尾的情况。）

这些技术可适用于大小为128或更大的范围，而不需额外的指令。例如，为了寻找取值在0到137（0x89）范围的最左字节的下标，可以简单地将上面搜索0到9之间的值的代码中的y=y|x改成y=y&x。

类似地，将上面的搜索取值范围在0x41到0x5A之间的最左字节的代码中的y=y|d改成y=y&d，就可以搜索取值范围在0x41到0xDA之间的最左字节。

6.2 寻找第一个给定长度的1位串

这里的问题就是在寄存器中寻找字的第一个给定长度n或更长的1位串，返回这个1位串的位置，如果不存在这样的串就要给出特殊的标记。它的变形有：只回答是/否标记，以及为第一个长度正好为n的1位串定位。这一问题可以应用于磁盘分配程序，特别是磁盘压缩（重新排列磁盘上的数据，使得每个文件的所有存储块都是连续的）。这一问题是Albert Chang向我建议的，他指出这是一个运用前导0数目指令的问题。

在此，我们假设前导0数目指令或这个函数的适当子例程是可用的。 96

立即能够想到的一个算法是：首先计算前导0的数目，再根据所得的前导0数目做左移位来略过这些前导0。接着通过进行1/0转换和前导0计数来计算前导1的数目。如果这个字足够长，我们的工作就完成了。否则，根据得到的前导0数目做左移位，再从开始重复。这一算法可以如下编码。如果找到了长度为n的连续1位，这个算法用0到31的数字返回最左侧的这样的序列的最左1位的位置。否则，它返回32作为“没有找到”的标识。

```
int ffstr1(unsigned x, int n) {
   int k, p;

   p = 0;                   // Initialize position to return.
   while (x != 0) {
      k = nlz(x);           // Skip over initial 0's
      x = x << k;           // (if any).
      p = p + k;
      k = nlz(~x);          // Count first/next group of 1's.
      if (k >= n)           // If enough,
         return p;          // return.
      x = x << k;           // Not enough 1's, skip over
      p = p + k;            // them.
   }
   return 32;
}
```

如果预期这个算法中的循环不会执行太多次，例如，如果x中有很长的1或0序列的话，那

么这个算法还是合理的。在磁盘分配应用中，这个算法有非常好的期望结果。然而，其最坏情况下的执行时间非常差；例如，当x=0x5555 5555且$n \geq 2$时，大约需要执行178个完全RISC指令。

有一个算法，其最坏情况下的执行时间稍好一些，它基于一系列左移位和与指令。为了弄清它是如何工作的，考虑在一个32位字x中搜索长度大于等于8的连续1位串。这可以通过如下所示的方法实现：

$$x \leftarrow x \,\&\, (x \ll 1)$$

$$x \leftarrow x \,\&\, (x \ll 2)$$

$$x \leftarrow x \,\&\, (x \ll 4)$$

在第一个赋值后，x中的1位给出长度为2的1位串的开始位置。第二个赋值后，x的1位给出长度为4（一个长度为2的1位串后面再跟着一个长度为2的1位串）的1位串的开始位置。第
97 三个赋值后，x中的1位给出长度为8的1位串的开始位置。在这个字中执行前导0数目指令，给出第一个长度大于等于8的连续1位串的位置，如果不存在这样的串，则结果是32。

为了开发对任意长度n（取值1到32）都适用的算法，我们从不同的角度观察这一算法。首先，注意到上面的三个赋值可以按任意顺序执行。把顺序颠倒将更方便。为了说明一般方法，考虑n=10的情况：

$$x_1 \leftarrow x \;\&\, (x \ll 5)$$

$$x_2 \leftarrow x_1 \,\&\, (x_1 \ll 2)$$

$$x_3 \leftarrow x_2 \,\&\, (x_2 \ll 1)$$

$$x_4 \leftarrow x_3 \,\&\, (x_3 \ll 1)$$

第一个语句是移位$n/2$个位。这个语句执行之后，问题就被简化成在x_1中寻找5个连续1位的串。可以通过将x_1左移位 $\lfloor 5/2 \rfloor$=2个位并与x_1做与计算，再搜索结果中长度为3(5−2)的连续1位串来完成这一工作。最后两个语句识别x_2中长度为3的串的位置。移位量的和总是$n-1$。图6-5给出了这一算法。对于1到32的n，算法执行3到36个完全RISC指令。

```
int ffstr1(unsigned x, int n) {
   int s;

   while (n > 1) {
      s = n >> 1;
      x = x & (x << s);
      n = n - s;
   }
   return nlz(x);
}
```

图6-5 寻找第一个连续n个1位的串，“移位和与”指令系列

如果n通常比较大，那么通过重复五次循环体来展开循环，同时省略测试n>1不是不合理。（对于32位计算机来说，5次总是足够的。）这给出了一个无分支的算法，它的运行时间是恒定的，执行20个指令（最后一个赋值可以省略）。对于较小的n，尽管比实际所需多执行3个赋值，但是，因为变量n的值1不变，这3个赋值不对x和n产生影响，所以这些额外的步骤不改变结

果。从被执行的指令数量上看，对于n > 5，展开循环的版本比循环版本要快捷。 98

可以多用6个指令（如果与和非可用，则多用4个指令）来寻找长度正好为n的连续1位串。对于每个长度大于等于n的连续1位串的开始位置，图6-5的算法计算出来的x的相应位都是1。因此，使用通过上述算法计算出来的x的最后值，x的最后值的每个孤立的1位，表达式

$$x \,\&\, \neg(x \overset{u}{\gg} 1) \,\&\, \neg(x \ll 1)$$

在相应位置的值是1，这表明原始的x有一个开始于此处的长度正好为n的连续1位串。

经过简单修改，这一算法也可用于寻找开始于特定位置的长度为n的连续位串。例如，为了寻找开始于字节边界的串，简单地对最后的x和0x8080 8080求与。

通过在开始时对x求补，或者把与运算改成或运算并在调用“nlz”之前对x求补，我们可以用这一算法寻找0位串。例如，下面是寻找第一个（最左）0字节的算法（对此问题的精确定义，参见6.1节）。

$$x \leftarrow x \mid (x \ll 4)$$
$$x \leftarrow x \mid (x \ll 2)$$
$$x \leftarrow x \mid (x \ll 1)$$
$$x \leftarrow \mathbf{0x7F7F7F7F} \mid x$$
$$p \leftarrow \mathrm{nlz}(\neg x) \overset{u}{\gg} 3$$

这一算法需要12个完全RISC指令（不如图6-2所给出的算法好，该算法需要8个指令）。 99

第7章　位和字节的重排列

7.1　位和字节的反转

“位反转”意指相对于寄存器的中心反射寄存器的内容，例如，

rev(0x01234567) = 0xE6A2C480

“字节反转”意指对寄存器中的4个字节做相似的反射。字节反转在DEC和Intel所用的“小端”格式与其他大多数计算机所用的“大端”格式之间的数据转换中是必要的操作。

通过交换相邻的单一位，然后再交换相邻的2位字段，以此类推，可以相当高效地实现位反转，如下所示[Aus1]。这5个赋值语句可以以任意顺序执行。

```
x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >>  1;
x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >>  2;
x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >>  4;
x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >>  8;
x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

通过使用更少的大常量，并用更直接的方法完成最后两个赋值语句，在大多数计算机上可以对上面的代码做一些改进，如图7-1所示（需要30个基本RISC指令，无分支）。

在这个代码中对x的最后一个赋值用9个基本RISC指令完成字节的反转。然而，如果计算机有循环移位指令，就可以利用下面的赋值在7个指令下完成字节的反转

$$x = ((x \,\&\, \mathbf{0x00FF00FF}) \overset{rot}{\gg} 8) \mid ((x \overset{rot}{\ll} 8) \,\&\, \mathbf{0x00FF00FF})$$

PowerPC可以仅用3个指令来完成字节的反转操作[Hay1]：8位左循环移位使4个字节中的两个字节移动到位，接着是两个“rlwimi”（立即左循环字跟掩码插入）指令。 101

```
unsigned rev(unsigned x) {
   x = (x & 0x55555555) <<  1 | (x >>  1) & 0x55555555;
   x = (x & 0x33333333) <<  2 | (x >>  2) & 0x33333333;
   x = (x & 0x0F0F0F0F) <<  4 | (x >>  4) & 0x0F0F0F0F;
   x = (x << 24) | ((x & 0xFF00) << 8) |
       ((x >> 8) & 0xFF00) | (x >> 24);
   return x;
}
```

图7-1　位的反转

1. 广义位反转

[GLS1]指出，下面这种称为“翻转（flip）”的广义位反转，完全可以包含在计算机指令集中：

```
if (k &  1) x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >>  1;
if (k &  2) x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >>  2;
if (k &  4) x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >>  4;
```

```
if (k &  8) x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >>  8;
if (k & 16) x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >> 16;
```

(最后的两个与操作可以省略)。对于k=31，这一操作反转字中的位。对于k=24，它反转字中的字节。对于k=7，它反转每个字节中的位，而不改变各字节的位置。对于k=16，它交换字中的左右两个半字，等等。一般说来，它把位置m上的位传送到位置$m \oplus k$上。可以用与通常的实现循环移位器类似的方法在硬件上实现这一算法。(需要MUX的五个阶段，每个阶段都受控于移位量k的某个位。)

2. 新颖的位反转方法

[HAK]第167条包含一些反转6位、7位和8位整数的相当深奥的表达式。尽管这些表达式是为36位计算机设计的，但是对6位整数的反转在32位计算机上也能运行。那些对7位和8位整数的反转可以在64位计算机上运行。这些表达式如下所示:

6位: remu((**x** * **0x00082082**) & **0x01122408**, **255**)

7位: remu((**x** * **0x40100401**) & **0x4422 1008**, **255**)

8位: remu((**x** * **0x2 02020202**) & **0x108 84422010**, **1023**)

102 所有这些表达式的结果都是一个“干净(clean)”的整数: 右对齐，不设置无用高阶位。

对于上述每种情况，都可以用“rem”或“ mod”取代“remu”函数，因为“remu”的参数是正的。这个求余函数简单地求以256或1024为底的数的各个数字之和，与弃九法(casting out nine)非常类似。因此，可以用一个乘指令和一个右移位指令替换它。例如，上述的6位反转公式在32位计算机上还有如下的表达式(乘必须是模2^{32}的):

$$t \leftarrow (x * \mathbf{0x00082082}) \,\&\, \mathbf{0x01122408}$$

$$(t * \mathbf{0x01010101}) \overset{u}{\gg} 24$$

这些公式的有效性是有限的，因为它们包含一个求余操作(至少需要20个周期)和/或一些乘法，并且需要装入较大的常量。上面表达式的执行需要10个基本RISC指令，其中2个是乘指令，在今天的RISC计算机上，它大约需要20个周期。与此相比，用于反转6位整数的图7-1所给的改进代码大约需要15个指令，并且根据计算机中指令级并行的数量，需要9到15个周期。然而，这些技巧给我们带来紧凑的代码。下面几项技术对32位计算机可能更有用。它们包含对[HAK]思想的双重应用，把[HAK]的技术扩展到32位计算机上的8位和9位整数的反转上。

下面是8位整数的反转公式:

$$s \leftarrow (x * \mathbf{0x02020202}) \,\&\, \mathbf{0x84422010}$$

$$t \leftarrow (x * 8) \,\&\, \mathbf{0x00000420}$$

$$\text{remu}(s + t, \mathbf{1023})$$

这里不能把“remu”改成乘和移位。(要想弄明白为什么，必须把结果计算出来，观察其位组合的模式。)

下面有一个与上述类似的反转8位整数的公式，这个公式很有趣，因为可以对它进行很大程度的简化:

$$s \leftarrow (x * \mathbf{0x00020202}) \,\&\, \mathbf{0x01044010}$$

$$t \leftarrow (x * \mathbf{0x00080808}) \,\&\, \mathbf{0x02088020}$$

$$\mathrm{remu}(s + t, \mathbf{4095})$$

之所以能够对它进行简化，是因为第二个积恰好是第一个积的左移位，可以用一个（移位）指令从第二个掩码生成最后的掩码，可以用乘和移位取代求余指令。它可以简化为14个基本RISC指令，其中2个是乘指令。 103

$$u \leftarrow x * \mathbf{0x00020202}$$

$$m \leftarrow \mathbf{0x01044010}$$

$$s \leftarrow u \,\&\, m$$

$$t \leftarrow (u \ll 2) \,\&\, (m \ll 1)$$

$$(\mathbf{0x01001001} * (s + t)) \overset{u}{\gg} 24$$

下面是反转9位整数的公式：

$$s \leftarrow (x * \mathbf{0x01001001}) \,\&\, \mathbf{0x84108010}$$

$$t \leftarrow (x * \mathbf{0x00040040}) \,\&\, \mathbf{0x00841080}$$

$$\mathrm{remu}(s + t, \mathbf{1023})$$

可以回避第二个乘法，因为第二个积等于第一个积右移位6个位置。最后一个掩码等于第二个掩码右移位8个位置。做这样的简化，得到需要12个基本RISC指令的公式，包括1个乘和1个求余指令。求余指令必须是无符号的并且不能改成乘和移位。

对这些技巧有研究的读者能够设计出其他的位置换操作的类似公式。作为一个简单的、人为的例子，假设希望从一个8位量中每隔一位提取一位，并将这四位压缩到右侧。也就是说，我们希望的变换是：

```
0000 0000 0000 0000 0000 0000 abcd efgh ==>
0000 0000 0000 0000 0000 0000 0000 bdfh
```

这一变换可以用下述公式实现：

$$t \leftarrow (x * \mathbf{0x01010101}) \,\&\, \mathbf{0x40100401}$$

$$(t * \mathbf{0x08040201}) \overset{u}{\gg} 27$$

在大多数计算机上，实现所有这些操作的最实用的方法就是编写一个1字节（或9位字段）整数的索引表。

3. 递增反转整数

快速傅立叶变换（FFT）算法在一个循环中使用整数i和它的位反转rev(i)，每次循环对i递增1[PB]。简单直接的编码在每次循环中递增i，然后计算rev(i)。对于较小的整数，通过查表来计算rev(i)快捷而实用。然而，对于大整数，查表法不实用，而且正如我们所看到的，计算rev(i)需要29个指令。 104

如果查表法不可用，那么，同时维护i和它的位反转形式并在每次循环中递增二者的做法也许更有效。这一做法引发一个问题，就是怎样递增以反转形式存放于寄存器中的整数最好。

为了说明这一点，假设在一台4位计算机上，我们希望依次求出（以十六进制形式表示的）下列值:

0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F

在FFT算法中，i和它的反转形式都是特定位长的数，位长几乎一定小于32，并且在寄存器中这些数是右对齐的。然而，我们假设i是32位整数。在反转的32位整数上加1之后，经过适当位数的右移位将使结果对FFT算法可用（i和rev(i)用于索引内存中的数组）。

递增反转整数的简明方法是，从左开始扫描第一个0位，将其设置为1，再将其左边的所有位都设置为0。这一做法的一种编码方式是:

```
unsigned x, m;

m = 0x80000000;
x = x ^ m;
if ((int)x >= 0) {
   do {
      m = m >> 1;
      x = x ^ m;
   } while (x < m);
}
```

如果x开始于0位，那么上面的代码在3个基本RISC指令之下完成，另外每次循环额外需要4个指令。因为x以0位开始的可能性是1/2，以（二进制）10开始的可能性是1/4，以此类推，故所需指令的平均数大约是:

$$3\times\frac{1}{2}+7\times\frac{1}{4}+11\times\frac{1}{8}+15\times\frac{1}{16}+\ldots$$
$$=4\times\frac{1}{2}+8\times\frac{1}{4}+12\times\frac{1}{8}+16\times\frac{1}{16}+\ldots-1$$
$$=4\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots\right)-1$$
$$=7$$

105

（我们在第二行进行了加1和减1操作，并用1/2+1/4+1/8+1/16+…的形式表示第1个1，这使这个级数与前面5.4节中讨论的级数类似。）然而在最坏情况下执行的命令数目相当大（131个指令）。

如果前导0数目指令是可用的，可以按如下方法实现反转后整数的加1:

首先执行: $s\leftarrow \mathrm{nlz}(\neg x)$

然后执行: $x\leftarrow x\oplus(\mathbf{0x8000\ 0000}\overset{s}{>>}s)$

或者: $x\leftarrow((x<<s)+\mathbf{0x8000\ 0000})\overset{u}{>>}s$

两个方法都需要5个完全RISC指令，为了从**0xFFFF FFFF**正确地回绕到**0**，我们需要移位是模64的。（在Intel x86上，因为移位是模32的，这些公式不能用。）

7.2 混洗位

对字中位的另一个重要的置换是“全混洗”（perfect shuffle）操作，这一操作在密码学中

有应用。“全混洗”有两种变形，分别称作“外”全混洗和“内”全混洗。它们二者运用类似于一副32张纸牌的全混洗方式，在一个字的两边相互交换位，两种混洗的不同之处在于允许哪张牌先落下来。在外全混洗中，外面（尾部）的位保留在外面的位置上，而在内全混洗中，第15位被传送到字的左端（第31位）。例如，对于如下的32位字（这里，每一个字母表示一个位）：

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
```

那么，外全混洗后，这个字变成

```
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

而内全混洗后，这个字变成

```
AaBb CcDd EeFf GgHh IiJj KkLl MmNn OoPp
```

假设字长W是2的幂，那么外全混洗操作可以使用基本RISC指令用$\log_2(W/2)$步完成，在这里每一步依次交换更小的分块的第二和第三个四分位块（quartile）[GLS1]。因此，一个32位字将做如下变换：

106

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
abcd efgh ABCD EFGH ijkl mnop IJKL MNOP
abcd ABCD efgh EFGH ijkl IJKL mnop MNOP
abAB cdCD efEF ghGH ijIJ klKL mnMN opOP
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

对这一混洗的简明编码是：

```
x = (x & 0x0000FF00) << 8 | (x >> 8) & 0x0000FF00 | x & 0xFF0000FF;
x = (x & 0x00F000F0) << 4 | (x >> 4) & 0x00F000F0 | x & 0xF00FF00F;
x = (x & 0x0C0C0C0C) << 2 | (x >> 2) & 0x0C0C0C0C | x & 0xC3C3C3C3;
x = (x & 0x22222222) << 1 | (x >> 1) & 0x22222222 | x & 0x99999999;
```

这一代码需要42个基本RISC指令。可以对这一代码进行修改，把指令数减到30，但是在一台无限制指令级并行的计算机上，执行周期将从17增加到21。修改的方法是使用异或方法来交换寄存器的两个字段。所有量都是无符号的：

```
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
```

以相反的顺序进行交换就很容易实现上述操作的逆操作，即外逆混洗：

```
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
```

使用上面所给出的两个混洗序列中任何一个的最后两步，就可以分别混洗每个字节的位。使用后三步则可以分别混洗每个半字的位，以此类推。同样的说明也适用于逆混洗(unshuffling)，但要使用前面的两步或三步。

为了得到内全混洗，我们在这些序列的前面加入交换寄存器左右半字的步骤：

```
x = (x >> 16) | (x << 16);
```

(或使用16个位置的循环)。可以通过在后面附加上行代码来对逆混洗序列做类似的修改。

107 把上面的转换变更为交换连续小块的第一和第四个四分位块，将生成内全混洗的位反转。

也许值得提出的特殊情况是字x的左半部的所有位都是0的情况。换句话说，我们想在x的右半部分的各位间插入0位，例如，把这个32位字

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

变换成

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P
```

可以将外全混洗算法简化成使用22个基本RISC指令完成上述任务。然而，下面的代码仅用19个指令就可完成此任务，在无限制指令级并行的计算机上也不损失执行时间（两个算法均需12个周期）。下面的代码不要求字x的左半部分最初就被明确初始化。

```
x = ((x & 0xFF00) << 8) | (x & 0x00FF);
x = ((x << 4) | x) & 0x0F0F0F0F;
x = ((x << 2) | x) & 0x33333333;
x = ((x << 1) | x) & 0x55555555;
```

类似地，作为这一“半混洗”操作的逆过程（压缩的特殊情况；参见7.4节），根据是否需要对奇数位置上的位清0的初始与操作，简化了的外全逆混洗代码可以在26到29个基本RISC指令下完成这一任务。然而，下面的代码只需要18或21个基本RISC指令来完成这一工作，并且在无限制指令级并行计算机上所用的执行时间更少（12到15个周期）。

```
x = x & 0x55555555;              // (If required.)
x = ((x >> 1) | x) & 0x33333333;
x = ((x >> 2) | x) & 0x0F0F0F0F;
x = ((x >> 4) | x) & 0x00FF00FF;
x = ((x >> 8) | x) & 0x0000FFFF;
```

7.3 转置位矩阵

矩阵*A*的转置矩阵是列为矩阵*A*的行、行为矩阵*A*的列的矩阵。这里，我们考虑计算位矩阵的转置矩阵的问题，位矩阵的元素是单一位，每八个位压缩到一个字节，而且列和行都开始于字节的边界。这看似简单的变换却令人惊讶地需要相当多的指令。

在大多数计算机上，装入和存储单独的位相当慢，这主要是因为这一工作需要提取并（更差的是）存储单独位的代码。一个更好的方法就是把矩阵分割成8 × 8的子矩阵，把每个8 × 8的子矩阵装入寄存器，计算寄存器中的子矩阵的转置矩阵，然后再将这个8 × 8矩阵的结

108 果存储在目标矩阵的适当位置。本节首先讨论8 × 8子矩阵的转置矩阵的计算问题。

是按行优先还是按列优先存储矩阵不是问题的关键；无论哪种情况，计算转置所需的操作都相同。假设我们的讨论是按行优先的顺序，使用8个装入字节指令将8 × 8子矩阵从源矩阵的列地址寻址并装入8个寄存器。也就是说，各装入字节指令所参考的地址的间隔是源矩阵的字节宽度的倍数。计算8 × 8子矩阵的转置矩阵之后，它被存储于目标矩阵的一列中，也就是说，用8个存储字节指令将这个子矩阵的转置矩阵放入适当地址，这些地址的间隔是目标矩阵的字节宽度的倍数（如果矩阵不是方阵，这个字节宽度与源矩阵的宽度不同）。因此，给定寄

存器a0、a1、a2、a3、a4、a5、a6、a7中右对齐的8个8位量，我们希望计算用于存储字节指令的寄存器b0、b1、b2、b3、b4、b5、b6、b7中右对齐的8个8位量。下面是转置的说明示例，其中，每个数字和字母代表一个位。注意，我们考虑的主对角线是从第0字节的第7位到第7字节的第0位。熟悉小端方式计算机的读者也许更习惯于把主对角线考虑成从第0字节的第0位到第7字节的第7位。

```
a0 = 0123 4567                b0 = 08go wEMU
a1 = 89ab cdef                b1 = 19hp xFNV
a2 = ghij klmn                b2 = 2aiq yGOW
a3 = opqr stuv      ==>       b3 = 3bjr zHPX
a4 = wxyz ABCD                b4 = 4cks AIQY
a5 = EFGH IJKL                b5 = 5dlt BJRZ
a6 = MNOP QRST                b6 = 6emu CKS$
a7 = UVWX YZ$.                b7 = 7fnv DLT.
```

解决这一问题的简明直观的代码是分别选择并放置每一个结果位，如下所示。其中，乘法和除法分别代表左移位和右移位。

```
b0 = (a0 & 128)     | (a1 & 128)/2   | (a2 & 128)/4   | (a3 & 128)/8 |
     (a4 & 128)/16  | (a5 & 128)/32  | (a6 & 128)/64  | (a7      )/128;
b1 = (a0 &  64)*2   | (a1 &  64)     | (a2 &  64)/2   | (a3 &  64)/4 |
     (a4 &  64)/8   | (a5 &  64)/16  | (a6 &  64)/32  | (a7 &  64)/64;
b2 = (a0 &  32)*4   | (a1 &  32)*2   | (a2 &  32)     | (a3 &  32)/2 |
     (a4 &  32)/4   | (a5 &  32)/8   | (a6 &  32)/16  | (a7 &  32)/32;
b3 = (a0 &  16)*8   | (a1 &  16)*4   | (a2 &  16)*2   | (a3 &  16)    |
     (a4 &  16)/2   | (a5 &  16)/4   | (a6 &  16)/8   | (a7 &  16)/16;
b4 = (a0 &   8)*16  | (a1 &   8)*8   | (a2 &   8)*4   | (a3 &   8)*2 |
     (a4 &   8)     | (a5 &   8)/2   | (a6 &   8)/4   | (a7 &   8)/8;
b5 = (a0 &   4)*32  | (a1 &   4)*16  | (a2 &   4)*8   | (a3 &   4)*4 |
     (a4 &   4)*2   | (a5 &   4)     | (a6 &   4)/2   | (a7 &   4)/4;
b6 = (a0 &   2)*64  | (a1 &   2)*32  | (a2 &   2)*16  | (a3 &   2)*8 |
     (a4 &   2)*4   | (a5 &   2)*2   | (a6 &   2)     | (a7 &   2)/2;
b7 = (a0       )*128| (a1 &   1)*64  | (a2 &   1)*32  | (a3 &   1)*16|
     (a4 &   1)*8   | (a5 &   1)*4   | (a6 &   1)*2   | (a7 &   1);
```

109

在大多数计算机上，这一代码的执行需要174个指令（62个与指令、56个移位指令和56个或指令）。当然，或指令也可以是与指令。在PowerPC上则可以用63个指令计算（7个传送寄存器指令和56个立即左循环字跟掩码插入指令）。我们没有把装入字节指令、存储字节指令以及寻址的代码计算在内。

尽管似乎没有解决这一问题的真正伟大的算法，但是下面将要描述的方法在基本RISC计算机上比简明方法至少快一倍。

首先，把8×8位矩阵当作是16个2×2位矩阵，转置这16个2×2位矩阵。第二，再把刚才得到的8×8位矩阵当作4个2×2子矩阵，这些子矩阵的元素是2×2位矩阵，并转置这4个2×2子矩阵。最后，再把刚才得到的8×8矩阵当作一个2×2矩阵，它的元素是4×4位矩阵，并转置这个2×2矩阵。这些变换如下所示。

```
0123 4567      082a 4c6e      08go 4cks      08go wEMU
89ab cdef      193b 5d7f      19hp 5dlt      19hp xFNV
ghij klmn      goiq ksmu      2aiq 6emu      2aiq yGOW
opqr stuv      hpjr ltnv      3bjr 7fnv      3bjr zHPX
          ==>            ==>            ==>
wxyz ABCD      wEyG AICK      wEMU AIQY      4cks AIQY
```

```
EFGH IJKL     xFzH BJDL     xFNV BJRZ     5dlt BJRZ
MNOP QRST     MUOW QYS$     yGOW CKS$     6emu CKS$
UVWX YZ$.     NVPX RZT.     zHPX DLT.     7fnv DLT.
```

不是在8个寄存器中的8个字节上完成这些步骤，而是先把8个字节中的4个字节压入一个寄存器，然后在两个寄存器中进行位交换，完成交换后再解压。完整的过程如图7-2所示。其中，参数A是$8m \times 8n$位源矩阵的8×8子矩阵的第一个字节的地址，参数B是$8n \times 8m$位目标矩阵中的8×8子矩阵的第一个字节的地址。也就是说，整个源矩阵是$8m \times n$字节，整个目标矩阵是$8n \times m$字节。

下面一行

```
t = (x ^ (x >> 7)) & 0x00AA00AA;  x = x ^ t ^ (t << 7);
```

的意义相当隐秘。它交换字x中（从右计算）第1位和第8位、第3位和第10位、第5位和第12位，以此类推；而不移动第0位、第2位，第4位，以此类推。这一交换使用了2.19节所述的异或位交换方法。第一轮交换前后的字x分别是

```
0123 4567 89ab cdef ghij klmn opqr stuv
082a 4c6e 193b 5d7f goiq ksmu hpjr ltnv
```

110

为了对这些矩阵转置方法做实际的比较，把上面所述的直观方法扩充成类似于图7-2所示的完整程序。使用GNU C编译器把两个程序编译到与基本RISC类似的目标计算机上。上面所述的那个直观方法需要219个指令，而图7-2的方法需要101个指令，包括所有装入指令、存储指令、寻址代码以及序言代码和收尾代码。图7-2的代码有一个适用于64位基本RISC计算机的变形，它大约需要85个指令。（在这种计算机中，x和y被放入同一寄存器中。）

```
void transpose8(unsigned char A[8], int m, int n,
                unsigned char B[8]) {
   unsigned x, y, t;

   // Load the array and pack it into x and y.

   x = (A[0]<<24)   | (A[m]<<16)   | (A[2*m]<<8) | A[3*m];
   y = (A[4*m]<<24) | (A[5*m]<<16) | (A[6*m]<<8) | A[7*m];

   t = (x ^ (x >> 7)) & 0x00AA00AA;  x = x ^ t ^ (t << 7);
   t = (y ^ (y >> 7)) & 0x00AA00AA;  y = y ^ t ^ (t << 7);

   t = (x ^ (x >>14)) & 0x0000CCCC;  x = x ^ t ^ (t <<14);
   t = (y ^ (y >>14)) & 0x0000CCCC;  y = y ^ t ^ (t <<14);

   t = (x & 0xF0F0F0F0) | ((y >> 4) & 0x0F0F0F0F);
   y = ((x << 4) & 0xF0F0F0F0) | (y & 0x0F0F0F0F);
   x = t;

   B[0]=x>>24;   B[n]=x>>16;   B[2*n]=x>>8;  B[3*n]=x;
   B[4*n]=y>>24; B[5*n]=y>>16; B[6*n]=y>>8;  B[7*n]=y;
}
```

图7-2 转置一个8×8位矩阵

基于被交换位组的长度，图7-2的算法按从细到粗的粒度运行。也可以按从粗到细的粒度

运行。为了实现这一工作，首先把8 × 8位矩阵当作一个元素为4 × 4位矩阵的2 × 2矩阵，转置这一2 × 2矩阵。然后把它的4个4 × 4子矩阵当作元素为2 × 2位矩阵的2 × 2矩阵，转置这4个2 × 2子矩阵的每一个子矩阵，以此类推。上述算法与图7-2的算法相同，不同的是执行位重排的三个语句组以相反顺序排列。

转置32 × 32位矩阵

8 × 8位矩阵的递归技巧同样可以用于更大的矩阵。对于32 × 32位矩阵，它需要五个阶段。 111

由于我们假设不能把整个32 × 32位矩阵装入通用寄存器空间，算法的细节部分与图7-2有相当大的不同。为了实现位交换，我们要寻找一个索引位矩阵中的适当字的紧凑算法。下述算法的最佳选择是按从粗到细的粒度来进行计算。

在第一个阶段，我们把32 × 32位矩阵当作4个16 × 16位矩阵，并做如下变换：

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix}$$

这里，A代表这个矩阵的前16个字的左半部分，B代表前16个字的右半部分，以此类推。显然，上面的变换可以用下面的交换来实现：

第0个字的右半部分与第16个字的左半部分交换，

第1个字的右半部分与第17个字的左半部分交换，

……

第15个字的右半部分与第31个字的左半部分交换。

为了对此进行编码，我们将使用一个取值范围在0到15的下标k。在一个由k控制的循环中，交换第k个字的右半部分与第k+16个字的左半部分。

在第二个阶段，把第一个阶段得到的矩阵当作16个8 × 8位矩阵，并对它做如下变换：

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \Rightarrow \begin{bmatrix} A & E & C & G \\ B & F & D & H \\ I & M & K & O \\ J & N & L & P \end{bmatrix}$$

这一变换可以用下面所示的交换来实现：

第0个字的0x00FF 00FF位与第8个字的0xFF00 FF00位交换，

第1个字的0x00FF 00FF位与第9个字的0xFF00 FF00位交换，以此类推。

这意味着将第0个字的第0到7位（最小有效8位）与第8个字的第8到15位交换，以此类推。在这些交换中，第一个字的下标依次是k=0、1、2、3、4、5、6、7、16、17、18、19、20、21、22、23。让k依次取这些值的方法是：

$$k' = (k + 9) \,\&\, \neg 8$$

在这个由k控制的循环中，第k个字的位与第k+8个字的位交换。 112

类似地，第三阶段做如下交换：

第0个字的0x0F0F 0F0F位与第4个字的0xF0F0 F0F0位交换，

第1个字的0x0F0F 0F0F位与第5个字的0xF0F0 F0F0位交换，以此类推。

在这些交换中，第一个字的下标依次是k=0、1、2、3、8、9、10、11、16、17、18、19、24、25、26、27。让k依次取这些值的方法是：

$$k' = (k + 5) \& \neg 4$$

在这个由k控制的循环中，第k个字的位与第k+4个字的位交换。

图7-3用C函数给出了这些步骤的相当紧凑的编码[GLS1]。外循环控制上述五个阶段，其中j的取值依次是16、8、4、2和1。同时，掩码m的取值依次是0x0000 FFFF、0x00FF 00FF、0x0F0F 0F0F、0x3333 3333和0x5555 5555（对此的编码是m=m^(m<<j)，这一编码是一个很好的小技巧。这一编码没有反转，这就是按从粗到细的粒度进行变换最有效的主要原因。）内循环控制k的取值如上所述。内循环体交换a[k]和经右移位j个位后的a[k+j]的某些位，这些位由掩码m所决定。掩码m对经右移位j个位后的a[k+j]的作用等同于m的补码对a[k+j]的作用。实现这一交换的代码使用了2.19节中所介绍的“三异或”技术。

```
void transpose32(unsigned A[32]) {
   int j, k;
   unsigned m, t;

   m = 0x0000FFFF;
   for (j = 16; j != 0; j = j >> 1, m = m ^ (m << j)) {
      for (k = 0; k < 32; k = (k + j + 1) & ~j) {
         t = (A[k] ^ (A[k+j] >> j)) & m;
         A[k] = A[k] ^ t;
         A[k+j] = A[k+j] ^ (t << j);
      }
   }
}
```

图7-3 转置32 × 32位矩阵的紧凑编码

用GNU C编译器将这一函数编译到与基本RISC非常类似的计算机时，这一代码需要31个指令，20个指令在内循环中，7个指令在外循环中、内循环外。因此，执行这一算法需要4+5 ×（7+16 × 20）=1639个指令。与此相比，如果通过16次调用图7-2的8 × 8转置程序来实现这一函数，假设16次调用是“按行展开”，那么将需要16 ×(101+5) =1696个指令。这包括每次调用函数时所需的5个指令（这可从编译后的代码得到）。因此，至少从表面上看，这两个方法在执行时间上大致相同。

113

另一方面，对于64位计算机，图7-3的代码很容易修改成转置64 × 64位矩阵，它将需要4+6 ×（7+32 × 20）=3886个指令。而通过执行64次8 × 8转置方法来做这一工作将需要64 ×(85+5）=5760个指令。

这一算法是原地工作（work in place）的，因此，如果用它来转置更大的矩阵的话，就需要传送32 × 32位子矩阵这一额外步骤。可以修改这一算法，通过把“j循环”的第一次或最后一次循环分离出来，让其把结果存储在其他区域，这样将结果放入源矩阵存储区域外的某个区域。

图7-3的算法所需要的指令中大约有一半是用来控制循环的，而且这个算法五次装入和存储整个矩阵。通过展开循环是否有可能减少这一开销呢？这是可能的，如果是在寻找极限速度，如果内存空间不是问题，如果计算机的信息提取跟得上大块的直线代码，而且，特别是

分支或装入的确耗时的话。展开后的算法，大部分工作是使用6个指令做位交换，共80次（5 × 16）。另外，这个算法需要32个装入指令来装入源矩阵，需要32个存储指令来存储结果，总共至少需要544个指令。

我们所用的GNU C编译器无法展开如此大的循环（内循环为16，外循环为7）。图7-4给出了手工展开循环的算法的概略。这一算法不是原地工作的，但是，如果需要，可以通过在调用它时传递相同参数来正确地实现原地执行。“交换”行的数目是80行。我们的GNU C编译器将这一算法编译成需要576个指令的基本RISC计算机代码（除了函数返回之外无分支）。包括序言代码和收尾代码。这一计算机没有多字存储和多字装入指令，但是可以使用双字存储和双字装入指令每次存储和恢复两个寄存器的内容。

```
#define swap(a0, a1, j, m) t = (a0 ^ (a1 >>j)) & m; \
                           a0 = a0 ^ t; \
                           a1 = a1 ^ (t << j);

void transpose32(unsigned A[32], unsigned B[32]) {
   unsigned m, t;
   unsigned a0, a1, a2, a3, a4, a5, a6, a7,
            a8, a9, a10, a11, a12, a13, a14, a15,
            a16, a17, a18, a19, a20, a21, a22, a23,
            a24, a25, a26, a27, a28, a29, a30, a31;

   a0  = A[ 0]; a1  = A[ 1]; a2  = A[ 2]; a3  = A[ 3];
   a4  = A[ 4]; a5  = A[ 5]; a6  = A[ 6]; a7  = A[ 7];
   ...
   a28 = A[28]; a29 = A[29]; a30 = A[30]; a31 = A[31];

   m = 0x0000FFFF;
   swap(a0,  a16, 16, m)
   swap(a1,  a17, 16, m)
   ...
   swap(a15, a31, 16, m)
   m = 0x00FF00FF;
   swap(a0,  a8,   8, m)
   swap(a1,  a9,   8, m)
   ...
   ...
   swap(a28, a29,  1, m)
   swap(a30, a31,  1, m)

   B[ 0] = a0;   B[ 1] = a1;   B[ 2] = a2;   B[ 3] = a3;
   B[ 4] = a4;   B[ 5] = a5;   B[ 6] = a6;   B[ 7] = a7;
   ...
   B[28] = a28;  B[29] = a29;  B[30] = a30;  B[31] = a31;
}
```

图7-4 转置32 × 32位矩阵的直线代码

如果计算机中有（左或右）循环移位指令，那么就有方法提高这个程序的效率。思路是，代替图7-4中需要6个指令的交换操作，使用不含移位的简单交换——这样的交换需要4个指令（将图中的swap宏中的移位省略）。

首先，右循环移位字A[16..31]（即，对应16 ≤ k ≤ 31的A[k]）16个位。其次，使用类似于图7-4的代码交换A[0]和A[16]、A[1]和A[17]等等的右半部分。再次，右循环移位A[0..8]和A[24..31]8个位，然后，利用与图7-4相似的代码，使用掩码0x00FF 00FF交换A[0]和A[8]、A[1]和A[9]等等字中的相应位。经过五个这样的阶段，得到一个不完全的转置矩阵。最后，
114 必须用左循环移位字A[1]一个位，左循环移位字A[2]两个位，以此类推（31个指令）。在此我们不给出代码，而是用下面的4 × 4位矩阵说明算法的步骤。

```
abcd      abcd      abij      abij      aeim      aeim
efgh  ==> efgh  ==> efmn  ==> nefm  ==> nbfj  ==> bfjn
ijkl      klij      klcd      klcd      kocg      cgko
mnop      opmn      opgh      hopg      hlpd      dhlp
```

115 图7-4的算法中，位重排部分需要480个指令（80次交换，每次需要6个指令）。用循环移位指令修改的算法需要80次交换，每次交换需要4个指令，五个阶段的第一步需要再加上80个循环移位指令，再加上最后的31个循环移位指令，总共需要431个指令。序言代码和收尾代码不变，所以，使用循环移位指令可以节省49个指令。

还有一个完全不同的转置位矩阵的方法：应用三个剪切变换[GLS1]。如果矩阵是$n \times n$，变换的步骤是，(1) 右循环移位第i行i个位，(2) 向上循环移位第j列（j+1）模n个位，(3) 右循环移位第i行（i+1）模n个位，(4) 沿着过中间点的水平轴反射这个矩阵。下面以4 × 4位矩阵为例加以说明：

```
abcd      abcd      hlpd      dhlp      aeim
efgh  ==> hefg  ==> kocg  ==> cgko  ==> bfjn
ijkl      klij      nbfj      bfjn      cgko
mnop      nopm      aeim      aeim      dhlp
```

这一方法与其他方法无法相比，因为步骤(2)的代价非常大。（为了在合理的代价之下实施步骤(2)，把所有需要向上循环移位$n/2$或更多位的列——从第$n/2-1$列到第$n-2$列——都向上循环移位$n/2$位，然后，再向上循环移位某些列$n/4$个位，以此类推。）步骤(1)和步骤(3)分别需要$n-1$条指令，如果结果能简单地被存储到适当位置的话，步骤(4)不需要指令。

如果用一种显然的方式（最上行存放在字的最大有效8位，以此类推），把一个8 × 8位矩阵存储到一个64位字中的话，那么矩阵转置操作等同于三个外全混洗或逆混洗[GLS1]。如果计算机中有作为单个指令的全混洗或逆混洗指令的话，用这一指令做矩阵转置是一个很好的方法。但是，在基本RISC计算机上这不是一个好方法。

7.4 压缩或广义提取

APL语言包含一种被称之为压缩的操作，写做B/V，其中B是一个布尔矢量，V是一个长度与B相同而元素任意的矢量。这一操作的结果是一个矢量，由对应于B中的1位的V中的元素组成，结果矢量的长度等于B中1位的数目。

这里，我们考虑对字的位做类似的操作。给定一个掩码m和一个字x，选择x中对应于掩码m的1位的位，并将这些位向右移动（“压缩”）。例如，如果下面是要压缩的字（这里的每一个字母都表示一个位）：

```
abcd efgh ijkl mnop qrst uvwx yzAB CDEF
```

而掩码是:

```
0000 1111 0011 0011 1010 1010 0101 0101
```

116

那么压缩的结果是:

```
0000 0000 0000 0000 efgh klop qsuw zBDF
```

也可以沿用大多数计算机中的提取指令的称呼，将这一操作称为广义提取。

我们感兴趣的是这一操作在最坏情况下用最少执行时间的算法，并提供图7-5所示的简单循环作为待改进的假想目标。这一算法的循环没有分支，在最坏情况下需要执行260个指令，包括子例程序言代码和收尾代码所需的指令。

```
unsigned compress(unsigned x, unsigned m) {
   unsigned r, s, b;    // Result, shift, mask bit.

   r = 0;
   s = 0;
   do {
      b = m & 1;
      r = r | ((x & b) << s);
      s = s + b;
      x = x >> 1;
      m = m >> 1;
   } while (m != 0);
   return r;
}
```

图7-5 压缩操作的简单循环算法

通过反复使用异或操作的并行前缀方法（参见5.2节）[GLS1]，可以对这一算法加以改进。我们用PP-XOR表示并行前缀操作。改进的基本思路是，首先识别出参数*x*中要向右移动奇数个位的位，把它们向右移动。（可以先用掩码与*x*取与来清除与压缩无关的位，以此来简化这一操作。）用同样的方法移动掩码。其次，识别出参数*x*中要向右移动2的奇数倍个位的位（如2、6、10等等），然后，再移动*x*和掩码的这些位。接下来，识别并移动那些要移动4的奇数倍个位的位，然后再识别并移动8的奇数倍个位的位，然后移动那些要移动16个位的位。

因为这一被认为是源于[GLS1]的算法理解起来相当困难，还因为这一算法还可以做一些令人惊讶的事情，所以我们仔细说明它的操作。假设输入是:

```
x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
m = 1000 1000 1110 0000 0000 1111 0101 0101
    1    1    111
    9    6    333                4444  3 2  1 0
```

117

其中，x中的每一个字母代表一个位（值为0或1）。掩码m的每一个1位下面的数字表示相对应的x的位要向右移动的位数。这个数目是m中这个位右边的0位的数目。如上面所提到那样，首先把x中的无关位清除掉，这给出:

```
x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F
```

我们的计划是，首先定出哪些位要向右移动奇数个位，再把这些位移动一个位。回想一下，PP-XOP操作在结果中产生这样一些1位：这一位及其右侧的1位的数目是奇数。我们希望

识别出右边的0位的数目是奇数的位。通过计算mk=~m<<1，并对结果实施PP-XOR，这样可以实现上述工作。结果是:

```
mk = 1110 1110 0011 1111 1110 0001 0101 0100
mp = 1010 0101 1110 1010 1010 0000 1100 1100
```

通过观察，可以看到mk识别出m中的右边是0位的位，而mp则从右边把这些位加起来再模2。因此，mp识别出m中右边有奇数个0位的位。

那些需要移动一个位的位是右边有奇数个0位（由mp识别）并且在原掩码中为1位的位。这就是mv=mp&m中的1位:

```
mv = 1000 0000 1110 0000 0000 0000 0100 0100
```

可以使用下面的赋值来移动m的这些位:

```
m = (m ^ mv) | (mv >> 1);
```

同时，可以使用下面的两个赋值来移动x的相应位:

```
t = x & mv;
x = (x ^ t) | (t >> 1);
```

(移动m的位比较简单，因为所有被选定的位都是1。)这里的异或把m 和x中已知是1的位改为0，而或把m和x中已知是0的位改成1。这些操作也可以都是异或，或者分别是减和加，通过mv把选定的位向右移动一个位后，结果是:

```
m = 0100 1000 0111 0000 0000 1111 0011 0011
x = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF
```

118

现在，我们必须为第二次循环准备一个掩码。在这一循环中，我们要识别出需要向右移动2的奇数倍个位的那些位。注意，mk&~mp识别出在原来的掩码m中右边为0位且有偶数个0位的那些位。这些特性可以共同运用到修正过的m上。(也就是说，mk识别修正过的m中的所有右边为0位且有偶数个0位的位。)这个量如果是从右侧开始用PP-XOR求和，那么它恰好识别出向右移动2的奇数倍（2、6、10、等等）个位的那些位。因此，要做的是，把这个量赋给mk，并在第二次循环中执行上面的步骤。mk的修正值是:

```
mk = 0100 1010 0001 0101 0100 0001 0001 0000
```

实现这一操作的完整C函数如图7-6所示。这一函数使用127个基本RISC指令，包括子例程序言代码和收尾代码。使用与上面讨论所用的同样的输入，图7-7给出了计算中在关键部位

119 的特定变量所取值的序列。注意，图7-6的算法有一个副产品：在m的最后一次赋值中，原来的m中的所有1位都被压缩到了右侧。

在64位基本RISC上，与图7-5的算法需要516个指令相比，图7-6的算法需要169个指令。

如果掩码m是一个常量，那么我们可以大幅度减少图7-6的算法所需要的指令数目。这在

120 以下两种情况下出现：(1) 在一个循环中调用compress(x,m)，在这一循环中m的值不是已知的，但它在循环中是常量，(2) m的值是已知的，compress的代码是事先生成的，可能是由编译器生成的。

```
unsigned compress(unsigned x, unsigned m) {
   unsigned mk, mp, mv, t;
   int i;

   x = x & m;              // Clear irrelevant bits.
   mk = ~m << 1;           // We will count 0's to right.

   for (i = 0; i < 5; i++) {
      mp = mk ^ (mk << 1);                  // Parallel prefix.
      mp = mp ^ (mp << 2);
      mp = mp ^ (mp << 4);
      mp = mp ^ (mp << 8);
      mp = mp ^ (mp << 16);
      mv = mp & m;                          // Bits to move.
      m = m ^ mv | (mv >> (1 << i));        // Compress m.
      t = x & mv;
      x = x ^ t | (t >> (1 << i));          // Compress x.
      mk = mk & ~mp;
   }
   return x;
}
```

图7-6 压缩操作的并行前缀方法

```
          x = abcd efgh ijkl mnop qrst uvwx yzAB CDEF
          m = 1000 1000 1110 0000 0000 1111 0101 0101
          x = a000 e000 ijk0 0000 0000 uvwx 0z0B 0D0F

i = 0,    mk = 1110 1110 0011 1111 1110 0001 0101 0100
After PP, mp = 1010 0101 1110 1010 1010 0000 1100 1100
          mv = 1000 0000 1110 0000 0000 0000 0100 0100
           m = 0100 1000 0111 0000 0000 1111 0011 0011
           x = 0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF

i = 1,    mk = 0100 1010 0001 0101 0100 0001 0001 0000
After PP, mp = 1100 0110 0000 1100 1100 0000 1111 0000
          mv = 0100 0000 0000 0000 0000 0000 0011 0000
           m = 0001 1000 0111 0000 0000 1111 0000 1111
           x = 000a e000 0ijk 0000 0000 uvwx 0000 zBDF

i = 2,    mk = 0000 1000 0001 0001 0000 0001 0000 0000
After PP, mp = 0000 0111 1111 0000 1111 1111 0000 0000
          mv = 0000 0000 0111 0000 0000 1111 0000 0000
           m = 0001 1000 0000 0111 0000 0000 1111 1111
           x = 000a e000 0000 0ijk 0000 0000 uvwx zBDF

i = 3,    mk = 0000 1000 0000 0001 0000 0000 0000 0000
After PP, mp = 0000 0111 1111 1111 0000 0000 0000 0000
          mv = 0000 0000 0000 0111 0000 0000 0000 0000
           m = 0001 1000 0000 0000 0000 0111 1111 1111
           x = 000a e000 0000 0000 0000 0ijk uvwx zBDF

i = 4,    mk = 0000 1000 0000 0000 0000 0000 0000 0000
After PP, mp = 1111 1000 0000 0000 0000 0000 0000 0000
          mv = 0001 1000 0000 0000 0000 0000 0000 0000
           m = 0000 0000 0000 0000 0001 1111 1111 1111
           x = 0000 0000 0000 0000 000a eijk uvwx zBDF
```

图7-7 压缩操作的并行前缀方法的运行示例

注意，在图7-6的循环中，赋给x的值除赋给x外不用在任何地方。而且x只依赖于它自身和变量mv的值。因此，可以修改循环的代码：删除所有对x的引用，分别用变量mv0，mv1，...mv4来存放mv的五个计算值。那么对于情况(1)，不引用x的函数可以放在原来调用compress(x,m)的循环之外，并在循环中放入下面的语句：

```
x = x & m;
t = x & mv0;   x = x ^ t | (t >> 1);
t = x & mv1;   x = x ^ t | (t >> 2);
t = x & mv2;   x = x ^ t | (t >> 4);
t = x & mv3;   x = x ^ t | (t >> 8);
t = x & mv4;   x = x ^ t | (t >> 16);
```

在循环中这一代码只需要21个指令（常量的装入可以在循环之外实现），这比图7-7所示的算法（需要127个指令）改进了很多。

在情况(2)中，m的值是已知的，所以可以做类似的工作，而且可以进一步简化。有可能5个掩码中有一个是0，如果是这样的话，可以省去上面所给的5个语句中的某一个。例如，如果不需要移动奇数个位的位时，掩码m1为0，如果不需要移动超过15个位的位时，那么m4=0，等等。

例如，设

```
m = 0101 0101 0101 0101 0101 0101 0101 0101
```

掩码的计算结果是：

```
mv0 = 0100 0100 0100 0100 0100 0100 0100 0100
mv1 = 0011 0000 0011 0000 0011 0000 0011 0000
mv2 = 0000 1111 0000 0000 0000 1111 0000 0000
mv3 = 0000 0000 1111 1111 0000 0000 0000 0000
mv4 = 0000 0000 0000 0000 0000 0000 0000 0000
```

因为最后的掩码是0，在已编译代码的情况下，这一压缩操作需要17个指令（不计掩码的装入）。这一结果不如7.2节最后所示的相关代码（在那里，不计掩码的情况下需要13个指令），因为
121 那里的操作充分利用了要变化的位是既定的这一事实。

1. 使用插入和提取指令

如果计算机中有插入指令，特别是识别目标寄存器中位字段的操作数是立即值时，那么在已编译的情况下，通常可以用插入指令来进行压缩操作，从而得到比上面所讨论的压缩方法用更少指令的算法。而且，插入指令不会占用寄存器来装入掩码。

首先，将目标寄存器初始化为0，然后，对于掩码m中的每个连续的1位组，将变量x右移位，使其右对齐到下一个字段的位置，用插入指令把x的位插入到目标寄存器的相应位置。这一操作需要$2n+1$个指令，其中n是掩码中字段的数目。最坏的情况需要33个指令，因为字段的最大数目是16（在1位与0位交替出现时有16个字段）。

插入方法使用较少指令的例子如m=0x0010 084A。使用这个掩码的压缩所需的移位量分别是1、2、4、8、16个位置。因此，使用并行前缀方法，这一工作整整需要21个指令，而插入方法（有5个字段）只需要11个指令。更极端的情况是m=0x8000 0000。这里有一个需要移动31个位置的位，用并行前缀方法需要21个指令，而用插入方法只需要3个指令，如果你没有被局限于特定方案的话，只需要1个指令（右移位31个位）。

如果掩码已知，也可以使用提取指令简单地用$3n-2$个指令实现压缩操作，其中n是掩码中字段的数目。

显然，对于掩码已知的情况，为压缩操作编译最优代码的问题是一个难题。

2. 左压缩

要把位向左侧压缩，显然可以反转参数x和掩码，向右压缩，再反转结果。另外一种方法是向右压缩，然后再左移位pop($\bar{m}$)个位。如果计算机中有位反转或种群计数指令的话，这些方法也许都会令人满意；但是如果没有这样的指令，那么可以很容易地将图7-6的算法改编成适用于这一问题的算法：简单地反转除表达式1<<i外的所有移位的方向（有8个移位需要改变）。

7.5 一般置换，分羊操作

要对字的位做一般置换，或任意其他的置换，一个核心问题就是如何表示这个置换。置换的表示不能太复杂，因为一个32位字的位置换有32!种，至少需要$\lceil \log_2(32!) \rceil = 118$位（即三个字加22位）来表明32!个置换中的一个。 122

表示置换的一个有趣的方法与7.4节所讨论的压缩操作有密切的关系[GLS1]。首先，直接的方法是简单地列出每个位要移动的目标位置。例如，为了表示左循环移位4个位的置换，第0位（最小有效位）移动到第4位，第1位移动到第5位……第31位移动到第3位。可以用32个5位索引矢量来表示这一置换：

```
00100
00101
 ...
11111
00000
00001
00010
00011
```

把这一表示看作是一个位矩阵，那么所要的置换表示就是这个位矩阵沿次对角线（off diagonal）的转置矩阵，这样第一行含有最小有效位，而且结果使用小端方式的位计数。我们将它放入数组p的五个32位字中：

```
p[0] = 1010 1010 1010 1010 1010 1010 1010 1010
p[1] = 1100 1100 1100 1100 1100 1100 1100 1100
p[2] = 0000 1111 0000 1111 0000 1111 0000 1111
p[3] = 0000 1111 1111 0000 0000 1111 1111 0000
p[4] = 0000 1111 1111 1111 1111 0000 0000 0000
```

p[0]中的每个位是x的相应位的移动目标的最小有效位，p[1]的每一位是下一个最小有效位，以此类推。这与前节中的用mv表示的掩码的编码类似，只是，在压缩算法中，mv作用于修正后的掩码，而不是原始掩码。

我们所需的压缩操作是把掩码中1位所标记的那些位向左压缩，同时把掩码中0位标记的位向右压缩[⊖]。有时把这一操作叫做“分羊”（sheep and goats）操作（SAG），或叫做“广义逆混洗”。可以用下式来计算它：

⊖ 如果使用大端方式进行位计数，那么向左压缩所有用0位标记的位，向右压缩所有用1位标记的位。

123

```
SAG(x, m) = compress_left(x, m) | compress(x, ~m)
```

以SAG为基本操作，同时使用上述的置换p，可以用如下的15步完成字x的位置换：

```
x    = SAG(x,    p[0]);
p[1] = SAG(p[1], p[0]);
p[2] = SAG(p[2], p[0]);
p[3] = SAG(p[3], p[0]);
p[4] = SAG(p[4], p[0]);

x    = SAG(x,    p[1]);
p[2] = SAG(p[2], p[1]);
p[3] = SAG(p[3], p[1]);
p[4] = SAG(p[4], p[1]);

x    = SAG(x,    p[2]);
p[3] = SAG(p[3], p[2]);
p[4] = SAG(p[4], p[2]);

x    = SAG(x,    p[3]);
p[4] = SAG(p[4], p[3]);

x    = SAG(x,    p[4]);
```

在这些步骤中，使用了SAG来实现稳定的二进制基数排序。数组p被当作32个对x的位排序的5位键。第一步，把x中所有对应于p[0]的1位的位向结果字的左半部分移动，所有对应于p[0]的0位的位向结果字的右半部分移动。除此之外，位的顺序不变（也就是说，排序是“稳定的”）。然后，对用于下一轮排序的键做同样的排序。第六行是根据键的次最小有效位对x排序，后面以此类推。

与压缩的情况类似，如果一个特定的置换p要作用于许多字x，那么可以通过预先计算上述的大部分步骤来产生一个相当节省的结果。置换数组可以改成：

```
p[1] = SAG(p[1], p[0]);
p[2] = SAG(SAG(p[2], p[0]), p[1]);
p[3] = SAG(SAG(SAG(p[3], p[0]), p[1]), p[2]);
p[4] = SAG(SAG(SAG(SAG(p[4], p[0]), p[1]), p[2]), p[3]);
```

然后使用下面的代码完成每一个置换：

```
x = SAG(x, p[0]);
x = SAG(x, p[1]);
x = SAG(x, p[2]);
x = SAG(x, p[3]);
x = SAG(x, p[4]);
```

124

对字的位做一般置换的更直接的方法（也许不太令人感兴趣）是用32个5位索引序列来表示置换。第k个索引是要移动到第k位的原字的位的位置。（这是一个“从……”（come from）列表，而SAG方法使用的是“至……”（go to）列表。）可以把每6个索引放入一个32位字中，因此需要6个字来装全部32个位索引。为了实现这一置换方法，可以考虑用硬件实现如下指令：

```
bitgather Rt,Rx,Ri,
```

其中，Rt是目标寄存器（也是源寄存器），寄存器Rx包含需要置换的那些位，寄存器Ri包含

六个5位索引（以及2个未用位）。指令操作为：

$$t \leftarrow (t \ll 6) \mid x_{i_0} x_{i_1} x_{i_2} x_{i_3} x_{i_4} x_{i_5}$$

换句话说，将目标寄存器的内容左移位6个位，从字x中选出6个位放入t的被腾出的六个位置。从x中选出的这6个位从左到右由字i的6个5位索引给出。索引中的位计数可以是小端计数方式、也可以是大端计数方式，这一操作的描述同时适合于这两种计算机。

要置换一个字，需使用6个这样的指令，所有指令中的Rt和Rx相同，但是索引寄存器的内容不同。在序列的第一个索引寄存器中只有索引i_4和i_5是有效的，因为其他4个索引所选出的位将从Rt的左端移出。

这一指令的实现很可能允许索引值重复，所以能够用这一指令来完成位置换外的其他工作。它可以用来在目标寄存器中重复任何一个选定的位任意次，而SAG操作缺乏这样的通用性。

把这一指令实现成快速（例如，1个周期）指令并非苛求。位选择回路由6个32：1的MUX回路组成。在目前的技术条件下，如果每个回路都用5个阶段的2：1 MUX回路构建的话（一共需要$6 \times 31 = 186$个MUX回路），这一指令要比一个32位加指令快[MD]。

位置换可应用于密码学，而与此密切相关的子字置换（即，置换字中字节）可应用于计算机图形学。这两个应用更有可能处理64位字或128位字，而不是32位字。经过直截了当的修改，SAG和*bitgather*方法可以用于以上各种大字长情况。

用DES算法对一条信息加密或解密需要大量的类置换映射。首先，要进行密钥生成，每个话路生成一个。这一操作包含17次类置换映射。第一条被称之为“第1置换筛选”的映射将一个64位量映射到一个56位量（它从密钥中选择56个无奇偶性位，并置换它们）。跟着是16次 125
从56位量到48位量的类置换映射，每次都使用被称为“第2置换筛选”的相同映射。

继密钥生成之后，对信息的每一个64位块做34次类置换操作。第一和最后的操作是互逆的64位置换。还有16次置换使用同一个映射将32位量映射到48位量。最后，还有16次使用同一映射的32位置换。总共有6个不同的映射，它们都是常量，并在[DES]中给出。

在“电子先锋基金会”（Electronic Frontier Foundation）于1998年使用特殊的硬件证明DES不可靠后，它就被摒弃了。美国国家标准和技术学会（NIST）认可使用Triple DES来暂时取代DES。Triple DES在每一个64位块上运行三次DES，每次使用不同的密钥（即，密钥的长度是192位，包括24个奇偶性位）。因此在加密和解密时，它需要的置换操作是DES的三倍。

然而，DES和Triple DES的“永久”替代品AES（之前的Rijndael算法[AES]）不涉及位级的置换。其中与置换最接近的操作是32位字的简单循环移位，移动的位数是8的倍数。被提议的及正在使用的其他加密方法一般都比DES涉及更少的位级置换。

比较上面讨论的两个置换方法，*bitgather*方法有以下优势：(1) 根据描述置换的原始数据生成索引字的准备工作更简单，(2) 硬件更简单，(3) 映射更通用。SAG方法有以下优势：(1) 用5个而不是6个指令做置换，(2) 它的指令格式只有两个源寄存器（这也许更适合于某些RISC的结构），(3) 可更好地扩展为置换双字量，(4)子字置换更有效。

[LSY] 讨论了第(3)项。做双字量的一般置换时，SAG方法使用两个SAG指令和若干基本RISC指令以及2个单字的全置换；*bitgather*方法使用三个单字的全置换，再加上若干基本RISC指令。其中，在置换中生成新量的预处理（这依赖于置换）所需的指令数不计算在内。我们把双字量的一般置换的具体方法留给读者。

关于第(4)项，例如，为了置换字中的四个字节，*bitgather*需要执行6个指令，这与用*bitgather*做一般的位置换所需要的指令一样多。但是，SAG只需要2个指令来做这一工作，而不是做一般的位置换所需的5个指令。即便在子字长不是2的幂时也可获得这一增效；它需要的步骤是$\lceil \log_2 n \rceil$，其中n是子字的数目，另外，不计两端的非参与位组。

[LSY]讨论了SAG和*bitgather*指令（分别称之为GPR和PPERM）、基于网络的其他置换指
126 令以及查表的置换方法。

7.6 重排列和索引变换

很多计算机字中的简单位重排列对应于更简单的关于位的坐标或索引变换[GLS1]。这些对应关系适合于任意一维数组元素的重排列，条件是数组元素的个数是2的整数幂。在编程应用中，当数组元素是计算机字或更大时，它们非常有用。

例如，当把结果存放于数组B时，长度为8的数组A的外全混洗由下列移动组成：

$$A_0 \to B_0; \quad A_1 \to B_2; \quad A_2 \to B_4; \quad A_3 \to B_6;$$
$$A_4 \to B_1; \quad A_5 \to B_3; \quad A_6 \to B_5; \quad A_7 \to B_7;$$

每一个B索引与向左循环移位1个位后的A索引相对应，这里的循环移位器是3位的。当然，通过向右循环移位每个索引可以实现外全逆混洗。一些类似的对应关系如表7-1所示。这里，n是数组元素的个数，“lsb”代表最小有效位，使用$\log_2 n$位的循环移位器实现索引的循环移位。

表7-1 重排列和索引转换

重排列	索引转换	
	数组索引、或大端方式的位计数	小端方式的位计数
逆置	取补	取补
位反转、或广义逆置	与一个常量取异或	与一个常量取异或
左循环移位k个位	减k（模n）	加k（模n）
右循环移位k个位	加k（模n）	减k（模n）
外全混洗	左循环移位1个位	右循环移位1个位
外全逆混洗	右循环移位1个位	左循环移位1个位
内全混洗	左循环移位1个位，然后对lsb取补	对lsb取补，然后右循环移位1个位
内全逆混洗	对lsb取补，然后右循环移位1个位	左循环移位1个位，然后对lsb取补
64位字中的8 × 8位矩阵的转置	（左或右）循环移位3个位	（左或右）循环移位3个位
FFT还原	反转位	反转位

127

第8章　乘　　法

8.1　多字乘法

多字乘法基本上是按传统小学中所使用的方法来完成的。然而，与使用部分乘积数组逐步展开的方法相比，把每个计算出部分积的行加到将存放最终积的行中的方法更加有效。

如果被乘数是m个字，乘数是n个字，那么无论带符号还是无符号，积占用$m+n$个（或更少的）字。

运用小学的乘法算法时，我们希望把每一个32位字当作单个数字。如果有计算两个32位整数的64位积的指令，那么乘法计算很容易。不幸的是，即使计算机有这样的指令，它也不是总能被大多数高级语言引用。事实上，多数现代RISC计算机没有这一指令，因为，高级语言不能引用它，而且也因此这一指令不常用。（另一个原因是，这一指令是给出两寄存器结果的极少的几个指令中的一个。）

我们的过程如图8-1所示，这一过程把半个字当作“数字”。参数w存放结果，u和v分别是乘数和被乘数。每个参数是半字的数组，而第一个半字（w[0]、u[0]和v[0]）是最小有效数字。这是一个“小端”的顺序。参数m和n分别是u和v中的半字的数目。

下面的图示有助于理解我们的过程。参数m和n之间没有特定的大小关系；二者中的任何一个都可能比另一个大。

$$
\begin{array}{r}
u_{m-1}u_{m-2}\ \cdots\ \cdots\ u_1\ u_0 \\
\times\ v_{n-1}\ \cdots\ v_1\ v_0 \\
\hline
w_{m+n-1}\ w_{m+n-2}\ \cdots\ \cdots\ \cdots\ w_1\ w_0
\end{array}
$$

我们的过程从算法M[Knu2, 4.3.1节]而来，但是这里是用C语言来描述的，而且改进为带符号乘法算法。注意，图8-1的上半部分对t的赋值不产生溢出，因为可能赋给t的最大值是$(2^{16}-1)^2+2(2^{16}-1)=2^{32}-1$。

无符号操作数的多字乘法最简单。事实上，如果省略“修正”步骤（三行说明与返回语句之间的行），图8-1的代码就执行无符号乘法。用三种方法可以将无符号乘法扩展到带符号乘法：

1）取每一个输入操作数的绝对值，执行无符号乘法，然后当输入操作数有不同的符号时取结果的负。

2）使用无符号基本乘法做乘法运算，当乘以一个高阶半字的情况时除外，此时使用“带符号×无符号”或“带符号×带符号”的乘法。

3）执行无符号乘法，然后采用某种方式修正结果。

第一个方法需要传递多至$m+n$个输入半字来计算它们的绝对值。或者，如果一个操作数是正的，另一个是负的，那么第一个方法需要传递多至$\max(m,n)+m+n$个输入半字来求负操作数和结果的补。也许更严重的是，这一算法会改变它的输入值（假定参数是通过地址传递的），

这使这一算法在某些应用中无法接受。可以选择给这些输入分配临时存储空间，或改变它们的值后再把它们改回原来的值。所有这些选择都没有吸引力。

```
void mulmns(unsigned short w[], unsigned short u[],
   unsigned short v[], int m, int n) {
   unsigned int k, t, b;
   int i, j;

   for (i = 0; i < m; i++)
      w[i] = 0;

   for (j = 0; j < n; j++) {
      k = 0;
      for (i = 0; i < m; i++) {
         t = u[i]*v[j] + w[i + j] + k;
         w[i + j] = t;            // (I.e., t & 0xFFFF).
         k = t >> 16;
      }
      w[j + m] = k;
   }

   // Now w[] has the unsigned product. Correct by
   // subtracting v*2**16m if u < 0, and
   // subtracting u*2**16n if v < 0.

   if ((short)u[m - 1] < 0) {
      b = 0;                        // Initialize borrow.
      for (j = 0; j < n; j++) {
         t = w[j + m] - v[j] - b;
         w[j + m] = t;
         b = t >> 31;
      }
   }
   if ((short)v[n - 1] < 0) {
      b = 0;
      for (i = 0; i < m; i++) {
         t = w[i + n] - u[i] - b;
         w[i + n] = t;
         b = t >> 31;
      }
   }
   return;
}
```

图8-1 多字整数乘法，带符号

第二个方法需要三种基本乘法（无符号×无符号，无符号×带符号，带符号×带符号），而且需要使用1或0将部分积向左进行符号扩展，从而使得每一个部分积需要更长的计算，并对总体计算产生影响。

我们选择第三个方法。为了了解这一方法的原理，设u和v表示两个相乘的带符号整数的值，而且设它们的长度分别是M和N位。那么图8-1的上半部错误地把u解释成一个无符号量，值为$u+2^M u_{M-1}$，这里u_{M-1}是u的符号位。也就是说，若u为负则$u_{M-1}=1$，否则$u_{M-1}=0$。同样，

程序也把v解释成值为$v+2^N v_{N-1}$的无符号数。

程序计算这些无符号数的积，也就是说，它计算：

$$(u + 2^M u_{M-1})(v + 2^N v_{N-1}) = uv + 2^M u_{M-1} v + 2^N v_{N-1} u + 2^{M+N} u_{M-1} v_{N-1}$$

为了得到希望的结果（uv），我们必须从无符号积中减去值$2^M u_{M-1} v+2^N v_{N-1} u$。没有必要减去$2^{M+N} u_{M-1} v_{N-1}$，因为我们知道结果可以用$M+N$位表示，所以没有必要去计算任何高于第$M+N-1$位的位。图8-1中“行注释”下的代码实现这两个减法。这两个减法最多传递$m+n$个半个字。

可能有人想向图8-1的程序传递全字整数的数组，即通过“跨越分界线”（lying across the interface）使用该算法。这时，算法在小端方式的计算机上可以正确运行，而在大端方式的计算机上则不能正确运行。如果我们用相反的顺序存储数组，即u[0]是最大有效半字（并对程序做相应修改），那么，“跨越分界线”程序可以在大端方式的计算机上正确运行，但不能在小端方式的计算机上正确运行。

130
~
131

8.2　64位积的高阶位部分

这里我们所考虑是两个32位整数积的高阶32位的问题。这一算法是基本RISC指令集中带符号积高阶位（mulhs）和无符号积高阶位（mulhu）指令的函数。

对于无符号乘法，图8-1上半部分的算法可以很好地工作。在特殊情况$m=n=2$下，对这一部分的算法做如下简单修改：展开循环，并把参数变成32位无符号整数。

对于带符号乘法，不需要编写图8-1下半部分的“修正部分”的代码。如果充分注意中间结果是带符号还是无符号（声明它们是带符号数时，右移位是符号传播移位）的话，就可以省略“修正部分”。最终算法如图8-2所示。对于无符号版本，简单地把所有的int改为unsigned。

无论是无符号版本还是带符号版本，图8-2的算法都需要16个基本RISC指令，其中4个是乘法指令。

```
int mulhs(int u, int v) {
   unsigned u0, v0, w0;
   int u1, v1, w1, w2, t;

   u0 = u & 0xFFFF;  u1 = u >> 16;
   v0 = v & 0xFFFF;  v1 = v >> 16;
   w0 = u0*v0;
   t  = u1*v0 + (w0 >> 16);
   w1 = t & 0xFFFF;
   w2 = t >> 16;
   w1 = u0*v1 + w1;
   return u1*v1 + w2 + (w1 >> 16);
}
```

图8-2　带符号乘高阶位

8.3　无符号积高阶位与带符号积高阶位间的转换

假设计算机能够计算两个无符号32位整数的64位积的高阶位部分，而我们希望对带符号

132 整数完成相应的操作。可以使用图8-2的算法，但是这需要4个乘法；[BGN]给出的算法比这一算法有效得多。

该算法是把Knuth的算法M从无符号乘法转换成带符号乘法的算法的特殊情况（见图8-1）。设x和y表示相乘的两个32位带符号整数。计算机将x解释成值为$x+2^{32}x_{31}$的无符号整数，其中x_{31}是x的最大有效位（也就是说，如果x是负数，那么x_{31}是整数1，否则是0）。类似地，y在无符号的解释下有值$y+2^{32}y_{31}$。

尽管我们想要的结果是xy的高阶32位，计算机却计算如下表达式：

$$(x + 2^{32}x_{31})(y + 2^{32}y_{31}) = xy + 2^{32}(x_{31}y + y_{31}x) + 2^{64}x_{31}y_{31}$$

要得到期望的结果，必须从这个量减去$2^{32}(x_{31}y+y_{31}x)+2^{64}x_{31}y_{31}$。因为我们知道结果可以是用64位表示，我们可以用模2^{64}来做算术。这就是说，我们可以很安全地忽略最后的项，并按如下所示的方法计算带符号积高阶位（7个基本RISC指令）。

$$\begin{aligned}
&p \leftarrow \text{mulhu}(x, y) && \text{// } \textit{multiply high unsigned} \text{ instruction.}\\
&t_1 \leftarrow (x \overset{s}{\gg} 31)\ \&\ y && \text{// } t_1 = x_{31}y.\\
&t_2 \leftarrow (y \overset{s}{\gg} 31)\ \&\ x && \text{// } t_2 = y_{31}x.\\
&p \leftarrow p - t_1 - t_2 && \text{// } p = \text{desired result.}
\end{aligned} \tag{8-1}$$

从带符号到无符号

很容易得到逆变换。逆变换的代码与上文(8-1)代码相同，只是第一个指令要变成带符号积高阶位，并且最后一个操作要改成$p \leftarrow p+t_1+t_2$。

8.4 常量乘法

显然，可以用一系列左移位指令和加指令来做常量乘法。例如，x乘以13（二进制为1101），可以编码如下：

$$\begin{aligned}
t_1 &\leftarrow x \ll 2\\
t_2 &\leftarrow x \ll 3\\
r &\leftarrow t_1 + t_2 + x
\end{aligned}$$

133 其中，r得到计算结果。

本节使用2的幂的乘法来表示左移位，所以上面的代码可以写成$r \leftarrow 8x+4x+x$，它表明在基本RISC和大多数计算机上，这一操作需要4个指令。

在此，我们想传达的是，这里有比表面看到的更多的东西。首先，除了计算给定常量的乘法所需要的移位和加指令的数量外，还有其他需要考虑的事项。为了说明，下面是乘以45（二进制为101101）的乘法的两个代码：

$$\begin{aligned}
t &\leftarrow 4x && t_1 \leftarrow 4x\\
r &\leftarrow x + t && t_2 \leftarrow 8x\\
t &\leftarrow 2t && t_3 \leftarrow 32x\\
r &\leftarrow r + t && r \leftarrow t_1 + x\\
t &\leftarrow 4t && t_3 \leftarrow t_3 + t_2\\
r &\leftarrow r + t && r \leftarrow r + t_3
\end{aligned}$$

左边的代码使用变量t来存放左移位后的x，移位量由乘数中的1位决定。每个移位的量通过前一个移位的量得到。这一代码有如下优点：

- 除了输入x和输出r之外，它只需要一个工作寄存器。
- 除了前两个之外，它只使用2地址指令。
- 移位量比较小。

当把此思路用于任意乘数时，也具有同样的性质。

右侧的代码首先做所有的移位操作，这里x是操作数。这一代码具有提高并行性的优点。在有足够的指令级并行的计算机上，右侧的代码需要3个周期，而左侧的代码在无限制并行的计算机上需要4个周期。

除了这些细节之外，寻找完成常量乘法所需要的最小操作数目不是一件容易的事，这里的“操作”是指典型计算机指令集合中的加和移位指令。以下我们假设这个指令集合是由加、减、位移量为任意常量的左移位以及取负指令组成的。我们假设指令的格式是三地址格式。然而，如果把指令局限于加指令（通过把一个数加到它自身，再把和加到它自身，以此类推来完成任意移位量的左移位），或者限制各指令只有一个参数，就如HPPA-RISC的移位和加指令那样，那么这一问题就不那么容易了。（这些移位和加指令先把寄存器的内容向左移位1个、2个或3个位，再把这一结果加到第二个寄存器中，最后再把结果放入第三个寄存器中。因此，它可以用一个可能很快的指令来乘3、5或9。）还假设我们只需要积的最小有效32位。 134

对上面暗示的基本二进制分解方案的第一个改进是：当乘数包含3个或更多的连续1位组时，使用减指令来缩短指令序列。例如，要乘以28（二进制为11100）时，可以计算$32x-4x$（3个指令），而不计算$16x+8x+4x$（5个指令）。在2的补码计算机上，即使$32x$的中间结果产生溢出、最后结果也是正确的且不产生溢出。

使用基本二进制分解方法（仅使用移位和加），乘以一个常量m的操作需要的指令数目是：

$$2\mathrm{pop}(m) - 1 - \delta$$

其中，如果m结束于1位（m是奇数），那么$\delta=1$，否则$\delta=0$。如果也可以用减指令，那么乘以一个常量m的操作所需的指令数目是：

$$4\mathrm{g}(m) + 2\mathrm{s}(m) - 1 - \delta$$

其中，$\mathrm{g}(m)$是m中两个或更多连续1位组的数目，$\mathrm{s}(m)$是m中孤立1位的数目，δ的含义与上式相同。

如果连续1位组的长度均为2，那么两种方法没有什么不同。

下一个改进是特殊处理被单个0位分开的组。例如，考虑$m=55$（二进制为110111）的情况。第一种改进方法通过计算$(64x-16x)+(8x-x)$来计算乘积，而这一计算需要6个指令。然而，把它当作$64x-8x-x$计算时只需要4个指令。类似地，乘以二进制数110111011时，我们可以计算$512x-64x-4x-x$（6个指令）。

上面的公式给出了变量x乘以任意给定的数m的操作数目的上界。下界可以通过m的长度得到，也就是说，它基于$n=\lfloor \log_2 m \rfloor + 1$。

定理　变量x乘以一个n位常量m（$m \geqslant 1$）的乘法可以用最多n个加、减和移位量为任意给定量的左移位指令来实现。

证明（对n应用归纳法）　乘以1的乘法可以用0个指令实现，所以定理对$n=1$成立。当$n>1$

时，如果m结束于0位，那么乘以m的乘法可以用乘以由m的左边$n-1$个位组成的数（也就是$m/2$）来实现。根据归纳假设，这最多需要$n-1$个指令，然后，对结果进行移位量为1的左移位操作。因此，总共最多需要n个指令。

如果m结束于二进制的01，那么可以通过x乘以由m的左边$n-2$位组成的数来实现mx，根据归纳假设，这最多需要$n-2$个指令，然后左移位结果两个位，再加上x。因此，总共最多需要n个指令。

135 如果m结束于二进制的11，那么考虑m结束于0011、0111、1011和1111的情况。设t是变量x乘以由m的左边$n-4$位组成的数的值。如果m结束于0011，那么$mx=16t+2x+x$，根据归纳假设，这一操作最多需要$(n-4)+4=n$个指令。如果m结束于0111，那么$mx=16t+8x-x$，同样最多需要n个指令。如果m结束于1111，那么$mx=16t+16x-x$，这最多需要n个指令。剩下的情况是m结束于1011。

很容易证明，如果m结束于001011、011011或者111011时，mx的计算最多需要n个指令。剩下的情况是m结束于101011。

持续这样的证明，“剩下的情况”总是101010…10101011的形式。最后总会达到m的长度，此时惟一剩下的情况是m为101010…10101011。这个n位数包含$n/2+1$个1位。通过前面的观察，它乘以x需要$2(n/2+1)-2=n$个指令。

因此，特别是在32位计算机上，利用上面的方法，乘以任何给定常量的乘法最多需要32个指令。通过观察，很容易看到，对于n是偶数的情况，n位数101010...101011需要n个指令；对于n是奇数的情况，n位数1010101...010110需要n个指令，所以下界是紧界。

到此为止，所讨论的方法用于计算或构造编译器可用的算法不是很难。但是，这样的算法不是总能生成最好的代码，因为有进一步改进的可能性。可以通过对乘数m或者在计算mx的过程中产生的中间结果进行因数分解来达到改进的目的。例如，再次考虑$m=45$（二进制为101101）的情况。上面所描述的方法需要6个指令。然而，将45因数分解为5×9给出一个4指令解：

$$t \leftarrow 4x + x$$
$$r \leftarrow 8t + t$$

可以把因数分解与加法的方法结合。例如，乘以106（二进制为1101010）的乘法通过加法方法需要7个指令，但是把106因数分解成$7\times15+1$将给出一个5指令解。

对于因数分解方法，据作者所知，乘以n位常量所需要的最大指令数目是一个悬而未决的问题。对于较大的n，这个最大值可能小于上面证明的n的上界。例如，m=0xAAAAAAAB时，在不用因数分解的情况下需要32个指令，但是把m的值写成$2\times5\times17\times257\times65537+1$的话，则得到一个10指令解。（然而，这个10指令解对大多数数字来说不具代表性。这一因数分解反映出了交替的1位和0位的单纯位模式）。

以上讨论显示，在这一看似简单的问题中包含着组合数字的思想。Knuth[Knu2,4.6.3节]讨论了与此紧密相关的运用最小数目的乘法计算a^m的问题。这一问题类似于仅用加法指令来
136 计算乘以m的问题。[Bern]描述了计算mx的编译器算法。

第9章　整数除法

9.1　预备知识

本章和下一章将给出几个“计算机整数除法”的算法和技巧。在数学公式中，我们用表达式x/y表示一般的有理数除法，而用$x \div y$表示带符号计算机整数除法（向零截取），用$x \overset{u}{\div} y$表示无符号计算机数除法。在C语言的代码中，x/y当然表示计算机除法，如果它的两个操作数都是无符号的，那么它就是无符号除，如果两个操作数都是带符号的，那么它就是带符号除。

除法运算是一个复杂的过程，含有除法的算法通常都不很精巧。甚至连如何定义整数除法都是一个值得研究的问题。大多数高级语言和大多数计算指令集将其定义为有理数结果的向零截取。这一定义以及其他两种可能的定义叙述如下：

		截取	模	地板
7÷3	=	2 rem 1	2 rem 1	2 rem 1
(-7)÷3	=	-2 rem -1	-3 rem 2	-3 rem 2
7÷(-3)	=	-2 rem 1	-2 rem 1	-3 rem -2
(-7)÷(-3)	=	2 rem -1	3 rem 2	2 rem -1

对于所有三种可能的定义，关系被除数＝商*除数＋余数都成立。在定义“模”（modulus）除法时，要求余数是非负数[⊖]。在定义“地板”（floor）除法时，要求商是有理数除法的结果的“地板”。对于除数是正的情况，模除法和地板除法等价。很少使用的第四种可能定义是向最接近的整数舍入。

模除法和地板除法的优越之处是大多数技巧可以简化。例如，除以2^n可以用带符号右移位n个位来取代，而且x除以2^n的余数可以用x和2^n-1的逻辑与给出。我想，模除法和地板除法通常可以给出想要的结果。例如，假设正在编写一个显示整数值函数曲线图的程序，函数的取值范围是最小整数$imin$到最大整数$imax$。你想把纵坐标的极限值设置为包含$imin$和$imax$的10的最小整数倍。那么，如果使用模除法或地板除法，则极限值分别是$(imin \div 10)*10$和$((imax+9) \div 10)*10$。如果使用传统除法，则必须做如下的计算：


```
if (imin >= 0) gmin = (imin/10)*10;
else           gmin = ((imin - 9)/10)*10;
if (imax >= 0) gmax = ((imax + 9)/10)*10;
else           gmax = (imax/10)*10;
```

除了模除法和地板除法的商比截取除法的商更有用外，而且我们认为更多的时候需要的是非负余数而不是负余数。

在模除法和地板除法之间作选择比较困难，因为只有当除数是负数的情况下，它们才是

⊖ 我知道自己可能因这一术语而受到责备，因为对于“模”蕴含着“非负”的含义这一点并未达成共识。Knuth[Knu1]的“mod”操作符是地板除法的余数，如果除数是负的，那么余数是负数或0。一些程序设计语言把“mod”作为截取除法的余数。然而，在数学里，有时候把“模”用于复数的大小，而且在同余理论中，模一般被假设为正的。

不同的，而除数是负数的情况并不常见。求助于现有的高级语言是没有用的，因为当操作数是带符号整数时，它们似乎都使用截取除法来处理x/y。少数高级语言给出了浮点数或有理数结果。余数的情况比较混乱。在Fortran 90中，MOD函数给出截取除法的余数，而MODULO函数给出地板除法的余数（可以是负的）。类似地，在Common Lisp和ADA中，REM是截取除法的余数，而MOD是地板除法的余数。在PL/I中，MOD总是非负的（它是模除法的余数）。在Pascal中，A mod B只有当B>0时才有定义，而且它取非负值（模除法和地板除法的余数）。

总之，即使我们知道我们多么想改变世界[⊖]，我们也不能改变世界，所以在以下的叙述中，我们使用通常的定义$x \div y$（截取除法）。

截取除法有一个很好的特性，那就是，对于$d \neq 0$，它满足：

$$(-n) + d = n + (-d) = -(n + d)$$

然而，当使用这个表达式进行程序变换时必须留意，因为如果n或d是最大负数时不能用32位来表示$-n$或$-d$。操作$(-2^{31}) \div (-1)$产生溢出（结果不能表示成2的补码形式的带符号量），而且在大多数计算机上，结果是无定义的，或者这一操作本身是不允许的。

带符号整数（截取）除法通过无符号除法与普通的有理数除法联系起来。也就是说，当带符号除法中的n和d都被解释成无符号整数时，满足公式(9-1)的上半部分。

$$n \div d = \begin{cases} \lfloor n/d \rfloor, & \text{if } d \neq 0, nd \geqslant 0 \\ \lceil n/d \rceil, & \text{if } d \neq 0, nd < 0 \end{cases} \tag{9-1}$$

138

在下面的讨论中，我们使用如下算术基本特性，在此我们不给出证明。参看[Knu]和[GKP]中对地板和天花板函数的有趣讨论。

定理D1 对实数x和整数k，有：

$$\lfloor x \rfloor = -\lceil -x \rceil \qquad \lceil x \rceil = -\lfloor -x \rfloor$$
$$x - 1 < \lfloor x \rfloor \leqslant x \qquad x \leqslant \lceil x \rceil < x + 1$$
$$\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1 \qquad \lceil x \rceil - 1 < x \leqslant \lceil x \rceil$$
$$x \geqslant k \Leftrightarrow \lfloor x \rfloor \geqslant k \qquad x \leqslant k \Leftrightarrow \lceil x \rceil \leqslant k$$
$$x > k \Rightarrow \lfloor x \rfloor \geqslant k \qquad x < k \Rightarrow \lceil x \rceil \leqslant k$$
$$x \leqslant k \Rightarrow \lfloor x \rfloor \leqslant k \Rightarrow x < k + 1 \qquad x \geqslant k \Rightarrow \lceil x \rceil \geqslant k \Rightarrow x > k - 1$$
$$x < k \Leftrightarrow \lfloor x \rfloor < k \qquad x > k \Leftrightarrow \lceil x \rceil > k$$

定理D2 对于整数n和d，如果$d>0$，有：

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d+1}{d} \right\rceil \quad \text{且} \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d-1}{d} \right\rfloor$$

如果$d<0$，有：

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lceil \frac{n-d-1}{d} \right\rceil \quad \text{且} \quad \left\lceil \frac{n}{d} \right\rceil = \left\lfloor \frac{n+d+1}{d} \right\rfloor$$

定理D3 对于实数x和整数d（$d \neq 0$），有：

⊖ 有人做过尝试。IBM的PL.8语言使用模除法，Knuth的MMIX计算机的除指令使用地板除法[MMIX]。

$$\lfloor \lfloor x \rfloor / d \rfloor = \lfloor x/d \rfloor \quad \text{且} \quad \lceil \lceil x \rceil / d \rceil = \lceil x/d \rceil$$

推论 对于实数a和b（$b \neq 0$）以及整数d（$d \neq 0$），有：

$$\left\lfloor \left\lfloor \frac{a}{b} \right\rfloor / d \right\rfloor = \left\lfloor \frac{a}{bd} \right\rfloor \quad \text{且} \quad \left\lceil \left\lceil \frac{a}{b} \right\rceil / d \right\rceil = \left\lceil \frac{a}{bd} \right\rceil$$

定理D4 对于整数n和d（$d \neq 0$），以及实数x，有：

如果$0 \leqslant x < \left|\frac{1}{d}\right|$，则$\left\lfloor \frac{n}{d} + x \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor$ 且，如果$-\left|\frac{1}{d}\right| < x \leqslant 0$，则$\left\lceil \frac{n}{d} + x \right\rceil = \left\lceil \frac{n}{d} \right\rceil$ 139

在下面的定理中，rem(n,d)表示n除以d的余数。对于负数d，定义rem(n,$-d$)=rem(n,d)，与截取除法及模除法的余数相同。我们不对n<0的情况使用rem(n,d)。因此，在我们的用法中，余数总是非负的。

定理D5 对于$n \geqslant 0$，$d \neq 0$，有：

$$\text{rem}(2n, d) = \begin{cases} 2\text{rem}(n, d) & \text{或} \\ 2\text{rem}(n, d) - |d|, \end{cases} \quad \text{且} \quad \text{rem}(2n+1, d) = \begin{cases} 2\text{rem}(n, d) + 1 & \text{或} \\ 2\text{rem}(n, d) - |d| + 1 \end{cases}$$

（无论哪种情况，余数的值都大于等于0，小于$|d|$。）

定理D6 对于$n \geqslant 0$，$d \neq 0$，有：

$$\text{rem}(2n, 2d) = 2\text{rem}(n, d)$$

定理D5和D6可以根据余数的基本定义得到证明，即对于$n \geqslant 0$和$d \neq 0$，存在整数q，使得：

$$n = qd + \text{rem}(n, d) \text{，其中 } 0 \leqslant \text{rem}(n, d) < |d|$$

（n和d可以不是整数，但是在这些定理中，我们只使用整数。）

9.2 多字除法

与多字乘法的情况一样，本质上说，我们可以用传统的小学除法来做多字除法。然而，细节部分却惊人的复杂。图9-1是Knuth的算法D[Knu2，4.3.1节]的C语言编码。除法的基础形式是32 $\frac{u}{\div}$ 16=>32。（实际上，这些基础除法操作的商最多是17位的。）

这一算法在处理它的输入和输出时，每次处理一个“半字”。当然，我们希望每次能处理一个全字，但是这样的算法需要能做64 $\frac{u}{\div}$ 32=>32除法的指令。在这里我们假设，或者计算机不具有这样的指令，或者这样的指令很难用于高级语言。尽管我们一般假设计算机有32 $\frac{u}{\div}$ 32=>32的除法，但对于这个问题只要有32 $\frac{u}{\div}$ 16=>16的除法就足够了。

因此，对于Knuth算法的这一实现，底b是65536。参见[Knu2]中对这一算法的解释。

被除数u和除数v按照“小端”的顺序，即u[0]和v[0]是最小有效数字。（算法在大端和小端方式的计算机上都能正确运行。）参数m和n分别是u和v中半字的数目（Knuth定义m为商的长度）。调用程序为商q提供空间，也可以也为余数r提供空间。给商提供的空间至少应该是m-n+1个半字，对余数提供的空间至少是n个半字。另外，可以对余数的地址赋NULL值来表 140

示不需要余数。

这一算法要求除数的最大有效数字v[n-1]是非零的。这简化了规范化的步骤，而且有助于确保调用程序给商提供足够的空间。代码检测v[n-1]是否为非零，也检测是否满足条件n≥1及m≥n。如果违反任何一个条件，那么代码返回一个出错代码（返回1）。

```
int divmnu(unsigned short q[], unsigned short r[],
     const unsigned short u[], const unsigned short v[],
     int m, int n) {

const unsigned b = 65536; // Number base (16 bits).
unsigned short *un, *vn;  // Normalized form of u, v.
unsigned qhat;            // Estimated quotient digit.
unsigned rhat;            // A remainder.
unsigned p;               // Product of two digits.
int s, i, j, t, k;

if (m < n || n <= 0 || v[n-1] == 0)
   return 1;              // Return if invalid param.

if (n == 1) {                              // Take care of
   k = 0;                                  // the case of a
   for (j = m - 1; j >= 0; j--) {          // single-digit
      q[j] = (k*b + u[j])/v[0];            // divisor here.
      k = (k*b + u[j]) - q[j]*v[0];
   }
   if (r != NULL) r[0] = k;
   return 0;
}

// Normalize by shifting v left just enough so that
// its high-order bit is on, and shift u left the
// same amount. We may have to append a high-order
// digit on the dividend; we do that unconditionally.

s = nlz(v[n-1]) - 16;          // 0 <= s <= 16.
vn = (unsigned short *)alloca(2*n);
for (i = n - 1; i > 0; i--)
   vn[i] = (v[i] << s) | (v[i-1] >> 16-s);
vn[0] = v[0] << s;

   un = (unsigned short *)alloca(2*(m + 1));
   un[m] = u[m-1] >> 16-s;
   for (i = m - 1; i > 0; i--)
      un[i] = (u[i] << s) | (u[i-1] >> 16-s);
   un[0] = u[0] << s;
   for (j = m - n; j >= 0; j--) {          // Main loop.
      // Compute estimate qhat of q[j].
      qhat = (un[j+n]*b + un[j+n-1])/vn[n-1];
      rhat = (un[j+n]*b + un[j+n-1]) - qhat*vn[n-1];
again:
```

图9-1 多字整数除法，无符号

```
      if (qhat >= b || qhat*vn[n-2] > b*rhat + un[j+n-2])
      { qhat = qhat - 1;
        rhat = rhat + vn[n-1];
        if (rhat < b) goto again;
      }

      // Multiply and subtract.
      k = 0;
      for (i = 0; i < n; i++) {
         p = qhat*vn[i];
         t = un[i+j] - k - (p & 0xFFFF);
         un[i+j] = t;
         k = (p >> 16) - (t >> 16);
      }
      t = un[j+n] - k;
      un[j+n] = t;

      q[j] = qhat;                  // Store quotient digit.
      if (t < 0) {                  // If we subtracted too
         q[j] = q[j] - 1;           // much, add back.
         k = 0;
         for (i = 0; i < n; i++) {
            t = un[i+j] + vn[i] + k;
            un[i+j] = t;
            k = t >> 16;
         }
         un[j+n] = un[j+n] + k;
      }
   } // End j.
   // If the caller wants the remainder, unnormalize
   // it and pass it back.
   if (r != NULL) {
      for (i = 0; i < n; i++)
         r[i] = (un[i] >> s) | (un[i+1] << 16-s);
   }
   return 0;
}
```

141
~
142

图9-1 （续）

这些检测之后，对于除数长度为1的简单情况，代码执行除法。从速度的角度考虑，我们单独处理这一情况；算法的其余部分需要除数的长度是2或更长。

如果除数的长度是2或更长，算法通过将除数向左移位以使它的最高位是1位来规范化除数。被除数向左移位相同的量，这样，通过这些移位，商不变。正如Knuth所说明的，这一规范化步骤使我们很容易精确地判断商的每个数字，所以这一步是必须的。我们使用前导0数目函数nlz(*x*)来决定移位量。

规范化步骤为被规范化的被除数和除数分配新空间。这样做的原因是，从调用程序的观点看，一般不希望输入参数被修改，还因为也许改变输入参数是不可能的，它们可能是存放于只读内存中的常量。另外，被除数也许需要额外的高阶数字。C语言的“alloca”函数是分配这一空间的理想函数。这一函数通常能非常有效地实现，只需要两三个内联指令来分配新

空间，而且不需要释放空间的指令。这一空间被分配在程序的栈区，当子例程返回时被自动释放。

在主循环中，每次循环生成商的一位数字，然后减小被除数，直到它变成余数。商的每位数字的评估值qhat，在循环中通过标签again进行循环求精之后，总是得到精确值或精确值加1。

下面的步骤是用除数乘以qhat，从当前的余数中减去积，这就像小学算术中所做的那样。如果余数是负的，需要对商的数字减1，并且需要重新乘和减或者简单地把除数加到余数上来修正余数。这一操作最多只能做一次，因为商的数字要么是精确值，要么比精确值多1。

最后，如果存放余数的地址是非空的，那么余数被放回到调用程序中。必须将余数向右移位，移位量是规范化移位量s。

很少需要执行“回加”（add back）步骤。为了明白这一点，注意，商的每个评估数字qhat的第一次计算是通过用当前余数的最大有效两位数字除以除数的最大有效数字来完成的。“again”循环中的步骤相当于把qhat修正为当前的余数的最大三位有效数字除以除数的最大两位有效数字（证明省略，读者可以用b=10的例子来说服自己）。注意，除数大于或等于b/2（由于规范化），并且被除数小于或等于b乘以除数（因为余数都小于除数）。

143 仅用三位被除数的数字和两位除数的数字对商进行评估可以精确到什么程度呢？因为已经做了规范化，商可以相当精确。为了更直观地了解这一点（这不是正式证明），考虑用这种方法，即，对底是10的算数评估u/v。可以看出，评估值总是大出（或精确）。因此，考虑评估值与精确值的误差的最坏情况：除数的两位数字截取，按照相对误差的意义，使除数减少得最多，并且，被除数的三位数字截取使被除数减少得最少（它是0），而且被除数足够大。考虑49900…0/5099…9，我们用499/50=9.98作为它的评估值，这时出现最坏的情况。真正的结果大约是499/51 ≈ 9.7843。0.1957的评估误差表明，评估出的商的数字和真商的数字都是相应比的地板函数，它们最大相差1，而且发生误差为1的可能性约为20%（假设商的数字是均匀分布的）。这反过来意味着“回加”步骤发生的可能性大约是20%。

将这一（非严格的）分析运用于一般的底b，得到评估商与真商的差最多是$2/b$的结果。对于b=65536，我们又得到评估商的数字与真商的数字最大差为1，而且发生这种情况的概率是2/65536 ≈ 0.00003的结果。因此大约只有商数字的0.003%需执行“回加”步骤。

需要回加步骤的例子如十进制的4500/501。底为65536时的类似例子如0x7FFF 8000 0000 0000/0x 8000 0000 0001。

我们不去对整个算法的运行时间做评估，但要注意的是，对于较大的m和n，执行时间是由乘/减循环决定的。好的编译器将把这一循环编译成大约16个基本RISC指令，其中有1个是乘指令。“for j”循环需要执行n次，乘/减循环需要执行$m-n+1$次，因此，这一部分程序的执行时间是$(15+mul)n(m-n+1)$个周期，其中，mul是两个16位变量相乘所需的时间。这一程序还要执行$m-n+1$个除指令和1个前导0数目指令。

带符号多字除法

我们不专门给出带符号多字除法的算法，而仅仅指出可以按如下方式修改无符号除法算法来实现这一目的：

1) 如果被除数是负的，就取它的负，对除数也做类似的操作。

2) 把被除数和除数转换成无符号表示。

3) 使用无符号多字除法算法。

4) 把商和余数转换成带符号表示。

5) 如果被除数和除数符号相反，取商的负。

6) 如果被除数是负的，取余数的负。

这些步骤有时需要加上或减去一个最大有效数字。例如，为简单起见，假设数是以256为底（每一个字节就是一个数字）的带符号整数，数字序列的高阶位是符号位。这与通常的2的补 144
码表示类似。那么，除数255的带符号表示是0x00FF，它在步骤2下被缩短到0xFF。类似地，如果步骤3得到的商是从1位开始的，那么为了得到正确的带符号量，一定要加上一个前导0字节。

9.3 从带符号除法到无符号短除法

“短除法”的意思是单字除以单字的除法（例如，32 ÷ 32=>32）。当操作数是整数时，C语言和大多数高级语言都提供了“/”操作符来表示这种形式的除法。C语言有带符号和无符号短除法，但是某些计算机在它们的指令表中只提供带符号除法。在这样的计算机上如何实现无符号除法呢？似乎没有什么好方法，但是在此我们给出某些可能性。

1. 使用带符号长除法

即使计算机有带符号长除法（64 ÷ 32=>32），无符号短除法也不像想象的那样简单。在IBM RS/6000计算机的XLC编译器中，无符号短除法$q \leftarrow (n \overset{u}{\div} d)$的实现如下：

$$\text{if } n \overset{u}{<} d \text{ then } q \leftarrow 0$$
$$\text{else if } d = 1 \text{ then } q \leftarrow n$$
$$\text{else if } d \leqslant 1 \text{ then } q \leftarrow 1$$
$$\text{else } q \leftarrow (0 \,\|\, n) \div d$$

其中，第3行实际是对$d \overset{u}{>} 2^{31}$的检测。如果此时d在代数意义下小于或等于1，那么因为d不等于1（从第2行可知），它一定小于或等于0。我们不考虑d=0的情况，只考虑那些有意义的情况，如果第3行的检测为**真**，那么d的符号位为1，即$d \overset{u}{>} 2^{31}$。因为，从第1行可知，$n \overset{u}{>} d$，而且n不超过$2^{32}-1$，所以$n \overset{u}{\div} d=1$。

第4行中的$(0 \| n) \div d$的意思是构造一个32个0位后面是32位量n的双字长整数，让它除以d。对d=1的检测（第2行）确保这一除法不会产生溢出（如果$n \overset{u}{>} 2^{31}$，除法会产生溢出，这时商无定义）。

通过共用第3行和第2行的比较[⊖]，上面的代码需要11个指令，其中3个是分支指令。在对 145
d=0执行除法时如果需要得到溢出中断的话，那么第三行可以改成“else if d<0 then $q \leftarrow 1$”，这在RS/6000计算机上给出一个12个指令的解。

很容易改变上面的代码，使得在通常情况下（$2 \overset{u}{<} d \overset{u}{<} 2^{31}$）不需通过那么多的检测（开始于if $d < 1$...），但是，代码的长度会稍稍增加。

2. 使用带符号短除法

如果带符号长除法不可用，但是带符号短除法是可用的，那么，可以通过把$n \overset{u}{\div} d$简化成$n,d<2^{31}$的情况，并使用计算机的除指令来实现$n \overset{u}{\div} d$。如果$d \overset{u}{>} 2^{31}$，那么$n \overset{u}{\div} d$只能是0或1，所以

⊖ 执行RS/6000计算机的比较指令设置表示小于、大于和等于的多个状态位。

这种情况很容易处理。于是，我们可以利用表达式$((n \overset{u}{\div} 2) \div d) \times 2$逼近$n \overset{u}{\div} d$的误差仅为0或1这一事实来简化被除数。这一思路导致下面的方法：

```
if d < 0 then if n ≤u d then q ← 0
  else q ← 1
else do
  q ← ((n ÷u 2) ÷ d) × 2
  r ← n − qd
  if r ≥u d then q ← q + 1
end
```

第1行中的检测$d<0$实际上是在检测$d \overset{u}{\geqslant} 2^{31}$是否成立。如果$d \overset{u}{\geqslant} 2^{31}$成立，那么最大的商是$(2^{32}-1) \div 2^{31}=1$，所以前两行计算正确的商。

第4行表示无符号右移位1个位、除、左移位1个位。显然，$n \overset{u}{\div} 2 \overset{u}{<} 2^{31}$，而且此时$d \overset{u}{<} 2^{31}$也成立，所以，可以使用计算机的带符号除指令。（如果$d=0$，会在此处发出溢出信号。）

在第4行计算出的评估值是：

$$q = \lfloor \lfloor n/2 \rfloor / d \rfloor \times 2 = \lfloor n/(2d) \rfloor \times 2 = \frac{n - \mathrm{rem}(n, 2d)}{d}$$

在这里，我们使用了定理D3的推论。第5行计算对应于评估商的余数。余数是：

$$r = n - \frac{n - \mathrm{rem}(n, 2d)}{d} \times d = \mathrm{rem}(n, 2d)$$

因此，$0 \leqslant r<2d$。如果$r<d$，那么q是正确的商。如果$r \geqslant d$，那么$1+q$就是正确的商（这一算法必须使用无符号比较，因为有$r \geqslant 2^{31}$的可能性）。

146

通过把0的立即装入到q的指令放在比较$n \overset{u}{<} d$之前，并把第2行的赋值$q \leftarrow 1$编码为分支到第6行的赋值$q \leftarrow q+1$，这一算法可以在大多数计算机上编写成14个指令的代码，其中4个是分支。也可以直接增加生成余数的代码：在第1行附加$r \leftarrow n$，在第2行附加$r \leftarrow n-d$，在第6行的“then”部分附加$r \leftarrow r-d$。（或者，以乘指令为代价，简单地把$r \leftarrow n-qd$附加到整个序列的最后。）

第1行和第2行可以改写成：

```
if n ≤u d then q ← 0
else if d < 0 then q ← 1
```

这可以使代码更紧凑一些，一共需要13个指令，其中3个是分支。但是，在通常情况下（满足$n>d$的较小数），这一代码需要执行更多的指令。

使用谓词表达式，这一程序可以写成：

```
if d < 0 then q ← (n ≥u d)
else do
  q ← ((n ÷u 2) ÷ d) × 2
  r ← n − qd
  q ← q + (r ≥u d)
end
```

如果有不用分支的谓词计算方法，那么这一代码节省2个分支。在Compaq Alpha计算机上，用一个指令（CMPULE）就可以计算这些谓词；在MIPS上，需要2个指令（SLTU，XORI）来计算。在大多数计算机上，通过使用2.11节给出的关于$x \overset{u}{<} y$的表达式并经过简化，这些谓词可以用4个指令来计算（如果计算机有完整的逻辑指令集合，那么需要3个指令），因为我们知道：在第1行$d_{31}=1$，在第5行$d_{31}=0$。第1行和第5行的谓词表达式可以分别简化成：

$$n \overset{u}{\geqslant} d = (n \,\&\, \neg(n-d)) \overset{u}{\gg} 31 \text{ 和}$$

$$r \overset{u}{\geqslant} d = (r \mid \neg(r-d)) \overset{u}{\gg} 31$$

我们可以通过当$d \overset{u}{\geqslant} 2^{31}$时强制令被除数为0来得到无分支代码。于是，除数就可以用于计算机的带符号除指令中，因为当除数被错误地解释为负数时，结果被设置为0，这在将进行修正的1的范围之内。对于大的被除数，我们的处理方法仍然是：在做除之前先将被除数向右移 147
位1个位、除之后再把商向左移位1个位。这给出了下面的程序（10个RISC指令）：

$$t \leftarrow d \overset{s}{\gg} 31$$

$$n' \leftarrow n \,\&\, \neg t$$

$$q \leftarrow ((n' \overset{u}{\div} 2) + d) \times 2$$

$$r \leftarrow n - qd$$

$$q \leftarrow q + (r \overset{u}{\geqslant} d)$$

9.4 无符号长除法

“长除法”的意思就是指双字除以单字的除法。对一台32位计算机而言，这一除法就是$64 \overset{u}{\div} 32 \Rightarrow 32$的除法，对于产生溢出的情况（包括除数为0的情况），对结果不作特殊规定。

有些32位计算机提供无符号长除法指令。然而，因为大多高级语言只能访问$32 \overset{u}{\div} 32 \Rightarrow 32$的除法，所以很少能利用无符号长除法的完整功能。因此，计算机的设计者可能选择只提供$32 \overset{u}{\div} 32$的除法指令，而且可能希望对实现$64 \overset{u}{\div} 32 \Rightarrow 32$除法的子例程的执行时间做出评估。这里我们给出两个长除法的算法。

1. 硬件“移位和减”算法

进行长除法的第一个设想是，考虑硬件所做的工作。通常使用的算法有两个，分别被称之为恢复和非恢复除法[H&P, A.2节; EL]。本质上，它们都是“移位和减”（shift-and-subtract）算法。在如下所示的恢复版本中，当减法给出负结果时，恢复步骤将减去的除数加回到被除数中。这里x、y和z都在32位寄存器中。初始的双字长被除数是$x \| y$，除数是z。我们需要一个1位寄存器来存放减法产生的溢出。

```
do i ← 1 to 32
  c ‖ x ‖ y ← 2(x ‖ y)                    // 左移位1位
  c ‖ x ← (c ‖ x) – (0b0 ‖ z)             // 减（33位）
  y0 ← ¬c                                 // 设置商的1个位
  if c then c ‖ x ← (c ‖ x) + (0b0 ‖ z)   // 恢复
end
```

148

结束时，商在寄存器y中，余数在寄存器x中。

在发生溢出的情况下，这个算法不能给出有用的结果。对于双字量x||y除以0，得到的商是x的1的补码，得到的余数是y。特别是，$0 \overset{u}{\div} 0 \Rightarrow 2^{32}-1$余0。其他溢出情况难以刻画。

对于非零除数，如果算法给出正确的商模2^{32}，并给出正确的余数，这也许很有用。然而，这样做的惟一方法似乎是要使表示c||x||y的寄存器具有97位，而且做64次循环。这是在做$62 \overset{u}{\div} 32 \Rightarrow 64$（译者注：原文如此，译者认为应该是$64 \overset{u}{\div} 32 \Rightarrow 64$。）的除法。减法仍旧是33位操作，但是额外的硬件和执行时间上的开销使这一改进没有太大价值。

这一算法在软件上很难精确实现，因为大多数计算机没有用于表示c||x的33位寄存器。然而，图9-2展示了在某种程度上反映了硬件算法的移位和减算法。

使用变量t使比较得到正确的结果。移位x||y之后，要做一个33位比较。如果（移位前）x的第一个位是1，那么这个33位量一定比除数（32位）大。在这种情况下，x|t的所有位均为1，所以比较给出正确的结果（真）。另一方面，如果x的第一位是0，那么只做32位的比较就足够了。

图9-2所示算法的代码需要321~385个基本RISC指令，这依赖于比较结果为真的频率。如果计算机有双字长左移位，那么移位操作只需要1个指令即可完成，而不是上面的4个指令。这样就可把执行时间减少到大约225~289个指令（作为循环控制，每次迭代允许2个指令）。

```
unsigned divlu(unsigned x, unsigned y, unsigned z) {
   // Divides (x || y) by z.
   int i;
   unsigned t;

   for (i = 1; i <= 32; i++) {
      t = (int)x >> 31;           // All 1's if x(31) = 1.
      x = (x << 1) | (y >> 31); // Shift x || y left
      y = y << 1;                 // one bit.
      if ((x | t) >= z) {
         x = x - z;
         y = y + 1;
      }
   }
   return y;                      // Remainder is x.
}
```

149

图9-2 无符号长除法，移位和减算法

可以通过令x=0来使用图9-2的算法完成$32 \overset{u}{\div} 32 \Rightarrow 32$除法。这一结果所带来的惟一简化是可以省略变量t，因为它总是0。

下面给出的是非恢复的硬件（无符号）除法算法。基本思路是，我们从记为c||x的33位量中减去除数z之后，如果结果是负的，没有必要把z加回去，而是在下一次循环执行加指令，而不是减指令。这是因为，加z（修正在前面的循环中减z的错误）、左移位、然后再减z，与加z等价（$2(u+z)-z=2u+z$）。这对硬件的好处是，在每次循环中只有一个加或减的操作，而且加法器可能是循环中最慢的回路[⊖]。如果余数是负的，那么需要在末尾对它进行调整。（不需要

⊖ 实际上，通过把减法的结果放在额外的寄存器中并且只当减（33位）的结果非负时将其写入x，可以回避恢复除法算法中的恢复步骤。但是，在某些实现中，这需要额外的寄存器和更多的运行时间。

对商做相应的调整。）

输入的被除数是双字量$x \| y$，除数是z。在结束时，商在寄存器y中，余数在寄存器x中。

```
c = 0
do i ← 1 to 32
  if c = 0 then do
    c || x || y ← 2(x || y)          // 左移位1位
    c || x ← (c || x) - (0b0 || z)   // 减除数
  end
  else do
    c || x || y ← 2(x || y)          // 左移位1位
    c || x ← (c || x) + (0b0 || z)   // 加除数
  end
  y0 ← ¬c                            // 设置商的1个位
end
if c = 1 then x ← x + z              // 如果为负就调整余数
```

这一编码似乎不能很好地适用于32位算法。

801迷你计算机（IBM创建的一台早期RISC实验计算机）有一个除法步骤（divide step）指令，它本质上是完成上面算法的循环体中的步骤。这一指令使用计算机的进位状态位来保存c，使用MQ（一个32位寄存器）来保存y。它的实现需要33位加法/减法器。801迷你计算机的除法步骤指令要比上面的循环稍稍复杂一些，因为它完成带符号除法，而且有一个溢出检测。使用这一指令，我们可以编写这样的除法子例程：它包含32个连续的除法步骤指令，后面跟着对商和余数的调整，以使余数带有所希望的符号。 150

2. 使用短除法

可以通过在图9-1所示的多字除法的算法中令m=4、n=2来得到$64 \stackrel{u}{\div} 32 \Rightarrow 32$的除法算法。当然还需要一些修改。参数应该是值传递的全字，而不是半字的数组。溢出条件不同；如果商不能包含在一个全字中的话，就会产生溢出。结果表明，对这一算法可以做很多简化。可以证明评估值qhat总是精确的；如果除数只由两个半字数字组成的话，这一评估是精确的。这意味着可以省略“回加”的步骤。如果展开图9-1的“主循环”及其内部的循环，那么还可以进一步做一些简化。

这些变换的结果如图9-3所示。被除数在u1和u0中，其中u1含有最大有效字。除数是参数v，商是函数的返回值。如果调用程序在参数r中提供了非空指针，那么函数将通过r所指的字返回余数。

在发生溢出时，算法返回等于最大无符号整数的余数，以此来作为发生溢出的标志。对于有效的除法操作，这是不可能出现的余数，因为余数必须小于除数。在发生溢出的情况下，程序还返回等于最大无符号整数的商，在某些情况下这个最大无符号整数足以表明余数不是我们所要的。

在给u32的赋值中有一个奇怪的表达式（-s>>31）。这是为了使当s=0时算法也能在有模32移位的计算机上（例如，Intel x86）正常运行。

通过均匀分布的随机整数的实验表明，在算法的每次执行中，执行“again”循环的概率是0.38。如果不需要余数，这给出了大约52个指令的执行时间。在这些指令当中，有1个前导0数目指令、2个除指令以及6.5个乘指令（不计乘b的指令，因为这是移位指令）。如果需要余数，则要再加上6个指令（包括存储r的指令），其中1个是乘指令。

divlu的带符号版本又如何呢？可能很难一步一步地修改图9-3中的代码来生成一个带符

```
unsigned divlu(unsigned u1, unsigned u0, unsigned v,
               unsigned *r)  {
   const unsigned b = 65536; // Number base (16 bits).
   unsigned un1, un0,        // Norm. dividend LSD's.
            vn1, vn0,        // Norm. divisor digits.
            q1, q0,          // Quotient digits.
            un32, un21, un10,// Dividend digit pairs.
            rhat;            // A remainder.
   int s;                    // Shift amount for norm.

   if (u1 >= v) {            // If overflow, set rem.
      if (r != NULL)         // to an impossible value,
         *r = 0xFFFFFFFF;    // and return the largest '
      return 0xFFFFFFFF;}    // possible quotient.

   s = nlz(v);               // 0 <= s <= 31.
   v = v << s;               // Normalize divisor.
   vn1 = v >> 16;            // Break divisor up into
   vn0 = v & 0xFFFF;         // two 16-bit digits.

   un32 = (u1 << s) | (u0 >> 32 - s) & (-s >> 31);
   un10 = u0 << s;           // Shift dividend left.

   un1 = un10 >> 16;         // Break right half of
   un0 = un10 & 0xFFFF;      // dividend into two digits.

   q1 = un32/vn1;            // Compute the first
   rhat = un32 - q1*vn1;     // quotient digit, q1.
again1:
   if (q1 >= b || q1*vn0 > b*rhat + un1) {
     q1 = q1 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again1;}

   un21 = un32*b + un1 - q1*v;  // Multiply and subtract.

   q0 = un21/vn1;            // Compute the second
   rhat = un21 - q0*vn1;     // quotient digit, q0.
again2:
   if (q0 >= b || q0*vn0 > b*rhat + un0) {
     q0 = q0 - 1;
     rhat = rhat + vn1;
     if (rhat < b) goto again2;}

   if (r != NULL)            // If remainder is wanted,
      *r = (un21*b + un0 - q0*v) >> s;     // return it.
   return q1*b + q0;
}
```

图9-3 无符号长除法，使用全字除法指令

号除法的算法。然而，可以通过取参数的绝对值、运行divlu，然后如果原始参数的符号不同则对结果求补，这样来实现带符号除法。对于诸如最大负数这样的极限值也不会出现问题，因为能用无符号整数正确表示任意带符号整数的绝对值。这一算法如图9-4所示。 151

```
int divls(int u1, unsigned u0, int v, int *r) {
   int q, uneg, vneg, diff, borrow;

   uneg = u1 >> 31;            // -1 if u < 0.
   if (uneg) {                 // Compute the absolute
      u0 = -u0;                // value of the dividend u.
      borrow = (u0 != 0);
      u1 = -u1 - borrow;}

   vneg = v >> 31;             // -1 if v < 0.
   v = (v ^ vneg) - vneg;      // Absolute value of v.

   if ((unsigned)u1 >= (unsigned)v) goto overflow;

   q = divlu(u1, u0, v, (unsigned *)r);

   diff = uneg ^ vneg;         // Negate q if signs of
   q = (q ^ diff) - diff;      // u and v differed.
   if (uneg && r != NULL)
      *r = -*r;

   if ((diff ^ q) < 0 && q != 0) {  // If overflow,
overflow:                       // set remainder
      if (r != NULL)            // to an impossible value,
         *r = 0x80000000;       // and return the largest
      q = 0x80000000;}          // possible neg. quotient.
   return q;
}
```

图9-4 带符号长除法，使用无符号长除法

对于带符号的情况，很难设计出检测溢出的真正好的代码。图9-4所示的算法给出了一个初步的检测，它与无符号长除法例程所使用的检测一样，确保$|u/v|<2^{32}$。经过这一检测后，只需保证商有正确的符号或是0即可。 152~153

第10章　整数常量除法

在许多计算机上，除法很耗时，如有可能应设法回避它。除法指令需要比加指令多20倍甚至更多执行时间的情况司空见惯，而且即使操作数很小，执行时间也同样很长。本章给出当除数是常量时回避除法指令的几个方法。

10.1　除以一个2的已知幂的带符号除法

很多人可能错误地认为，可以用除法的常用截取形式，把一个数带符号右移位k个位就是用2^k去除一个数[GLS2]。实际上的除法要比这复杂一些。下面的代码对$1 \le k \le 31$计算$q=n \div 2^k$[Hop]：

```
shrsi t,n,k-1       Form the integer
shri  t,t,32-k      2**k - 1 if n < 0, else 0.
add   t,n,t         Add it to n,
shrsi q,t,k         and shift right (signed).
```

这段代码是无分支的。对于常用的除以2（k=1）的情况，它也可以简化成3指令代码。然而，这要依赖于计算机能否在很短的时间内做较大移位量的移位。k=31的情况没有太大意义，因为在计算机中数2^{31}是不可表示的。不过，这个代码在k=31的情况下的确也可以生成正确解（其中，如果$n=-2^{31}$则$q=-1$，而对其他所有n，q=0）。

要除以-2^k，上面的代码可以加上一个取负指令。似乎没有更好的方法做这一除法。

除以2^k的更简明的代码是：

```
        bge   n,label       Branch if n >= 0.
        addi  n,n,2**k-1    Add 2**k - 1 to n,
label   shrsi n,n,k         and shift right (signed).
```

对于执行移位指令慢而执行分支指令快的计算机，这一代码是可取的。

PowerPC有一个非凡设计，可以加快除以2的幂的除法[GGS]。如果被移位的数是负数而且有一个以上的1位被移出的话，带符号右移位指令设置计算机的进位位。PowerPC还有一个把进位加到寄存器的指令，记为addze。这一指令使得除以任何一个2的（正）幂的除法都可用两个指令实现：


```
shrsi q,n,k
addze q,q
```

含有k个位置的一条shrsi指令能够实现与模除法及地板除法一致的除以2^k的带符号除法。这表明对计算机及HLL来说，这些除法中的一个也许比截取除法更合适。也就是说，模除法和地板除法比截取除法更适合shrsi，它允许编译器将表达式$n/2$转化成一个shrsi。另外，shrsi后面跟着一个neg（求负）指令可以实现除以-2^k的除法，这给我们一个提示，模除法可能是最好的。（然而，这个方法主要用于研究，它没有太大的实际意义，因为除以一个

负常量的除法是很少见的。）

10.2 除以一个2的已知幂的除法的带符号余数

如果同时需要$n \div 2^k$的商和余数的话，通过$r=q*2^k-n$计算余数r的方法最简单。计算出商q之后，这只需要两个指令。

```
shli   r,q,k
sub    r,r,n
```

只计算余数似乎大约需要4到5个指令。计算余数的一个方法就是利用上面所给出的除以2^k的带符号除法的4指令序列，后面再加上刚才给出的计算余数的两个指令。这导致有两个连续的可以用与指令取代的移位指令，这给出一个5指令解（如果$k=1$，则需要4个指令）：

```
shrsi  t,n,k-1        Form the integer
shri   t,t,32-k       2**k - 1 if n < 0, else 0.
add    t,n,t          Add it to n,
andi   t,t,-2**k      clear rightmost k bits,
sub    r,n,t          and subtract it from n.
```

另外一个方法基于下面的公式：

$$\mathrm{rem}(n, 2^k) = \begin{cases} n \,\&\, (2^k - 1), & n \geqslant 0 \\ -((-n) \,\&\, (2^k - 1)), & n < 0 \end{cases}$$

要使用这个公式，首先计算$t \leftarrow n \overset{s}{>>} 31$，然后计算

156

$$r \leftarrow ((\mathrm{abs}(n) \,\&\, (2^k - 1)) \oplus t) - t$$

（5个指令），或对$k=1$，因为$(-n)\&1=n\&1$，所以有

$$r \leftarrow ((n \,\&\, 1) \oplus t) - t$$

（4个指令）。如果计算机没有绝对值指令（这时，余数的计算将需要7个指令），那么这一方法对$k>1$的情况并不理想。

还有一个基于下面公式的方法：

$$\mathrm{rem}(n, 2^k) = \begin{cases} n \,\&\, (2^k - 1), & n \geqslant 0 \\ ((n + 2^k - 1) \,\&\, (2^k - 1)) - (2^k - 1), & n < 0 \end{cases}$$

这将给出

$$t \leftarrow (n \overset{s}{>>} k - 1) \overset{u}{>>} 32 - k$$

$$r \leftarrow ((n + t) \,\&\, (2^k - 1)) - t$$

（$k>1$时需要5个指令，$k=1$时需要4个指令）。

上面的方法对所有$1 \leqslant k \leqslant 31$都有效。

顺便提一下，如果带符号右移位指令不可用，那么可用下面的代码计算结果，结果是：当$n<0$时值是2^k-1，当$n \geqslant 0$时值是0。

$$t_1 \leftarrow n \overset{u}{>>} 31$$

$$r \leftarrow (t_1 \ll k) - t_1$$

这一代码只增加1个指令。

10.3 非2的幂的带符号除法和余数

这里的基本技巧就是，乘以类似于除数d的倒数的数，它近似于$2^{32}/d$，然后提取积的最左边32位。然而，细节部分要复杂得多，特别是对7这样的特定除数。

我们首先考虑几个特殊的例子，这些例子可以展示适用于一般方法的代码。我们标记寄存器如下：

n——输入整数（分子）
M——装入“魔术数”
t——临时寄存器
q——将存放商
r——将存放余数

157

1. 除以3

```
li    M,0x55555556  Load magic number, (2**32+2)/3.
mulhs q,M,n         q = floor(M*n/2**32).
shri  t,n,31        Add 1 to q if
add   q,q,t         n is negative.

muli  t,q,3         Compute remainder from
sub   r,n,t         r = n - q*3.
```

证明 因为两个32位整数的积总可以用64位表示，而且带符号乘高阶位（mulhs）操作给出这个64位积的高阶32位，所以带符号乘高阶位操作不会产生溢出。这等价于用64位积除以2^{32}，并取结果的地板，无论积是正的还是负的，这都是正确的。因此，对于$n \geq 0$，上面的代码计算：

$$q = \left\lfloor \frac{2^{32}+2}{3} \times \frac{n}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3\times 2^{32}} \right\rfloor$$

现在，对于$n<2^{31}$，因为$2^{31}-1$是最大可表示的正数，因此“误差”项$2n/(3\times 2^{32})$小于1/3（而且是非负的），所以根据定理D4（参见9.1节），我们有$q=\lfloor n/3 \rfloor$，这就是所需的结果（参见9.1节等式(9-1)）。

对于$n<0$，这里有一个把1加到商的运算，因此代码计算：

$$q = \left\lfloor \frac{2^{32}+2}{3} \times \frac{n}{2^{32}} \right\rfloor + 1 = \left\lfloor \frac{2^{32}n + 2n + 3\times 2^{32}}{3\times 2^{32}} \right\rfloor = \left\lceil \frac{2^{32}n + 2n + 1}{3\times 2^{32}} \right\rceil$$

这里我们使用了定理D2。因此有

$$q = \left\lceil \frac{n}{3} + \frac{2n+1}{3\times 2^{32}} \right\rceil$$

对于$-2^{31} \leq n \leq -1$，有

$$-\frac{1}{3} + \frac{1}{3\times 2^{32}} \leqslant \frac{2n+1}{3\times 2^{32}} \leqslant -\frac{1}{3\times 2^{32}}$$

误差项是非正的，而且大于-1/3，所以根据定理D4，$q=\lceil n/3 \rceil$，这是所需的结果（参见等式(9-1)）。

这证明了商是正确的。由于余数必须满足

$$n = qd + r$$

这里的乘以3不会产生溢出（因为$-2^{31}/3 \leq q \leq (2^{31}-1)/3$），而且因为减的结果一定在-2到+2之
158 间，所以减法也不能产生溢出。由上易知，余数是正确的。

可以用2个加，或1个移位和1个加来实现立即乘指令，前提是它们之一能够改进执行的时间。

在很多今天的RISC计算机上，上述的商的计算需要9或10个周期，而除指令可能需要大约20个周期。

2. 除以5

对于除以5的除法，我们也希望使用与除以3相同的代码，只不过乘数是$(2^{32}+4)/5$。不幸的是，误差项太大；对于$n \geq 2^{30}$，有1/5的时候计算结果相差1。然而，我们可以使用$(2^{33}+3)/5$作为乘数，再加上1个带符号右移位指令。代码是：

```
li    M,0x66666667  Load magic number, (2**33+3)/5.
mulhs q,M,n         q = floor(M*n/2**32).
shrsi q,q,1
shri  t,n,31        Add 1 to q if
add   q,q,t         n is negative.

muli  t,q,5         Compute remainder from
sub   r,n,t         r = n - q*5.
```

证明 mulhs产生64位积的最左32位，然后代码将其带符号（或“算术”）向右移1位。这等价于用积除以2^{33}再取这一结果的地板。因此，对于$n \geq 0$，代码计算

$$q = \left\lfloor \frac{2^{33}+3}{5} \times \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{3n}{5 \times 2^{33}} \right\rfloor$$

对于$0 \leq n < 2^{31}$，误差项$3n/5 \times 2^{33}$是非负的，而且小于1/5，所以根据定理D4，$q=\lfloor n/5 \rfloor$。

对于$n<0$，上面的代码计算

$$q = \left\lfloor \frac{2^{33}+3}{5} \times \frac{n}{2^{33}} \right\rfloor + 1 = \left\lceil \frac{n}{5} + \frac{3n+1}{5 \times 2^{33}} \right\rceil$$

误差项是非正的，而且大于-1/5，所以$q=\lceil n/5 \rceil$。

与除以3的情况相同，可以证明余数是正确的。

可以用1个移位量为2的左移位和1个加来实现立即乘。

3. 除以7

除以7的情况产生了一个新问题。乘数$(2^{32}+3)/7$和$(2^{33}+6)/7$给出的误差都太大。乘数
159 $(2^{34}+5)/7$可能有效，但是它太大、不能表示成32位带符号字。可以利用乘以$(2^{34}+5)/7-2^{32}$（一个负数）来代替乘以这个大数，然后，再用插入一个add修正这个积。代码是：

```
li    M,0x92492493  Magic num, (2**34+5)/7 - 2**32.
```

```
mulhs q,M,n          q = floor(M*n/2**32).
add   q,q,n          q = floor(M*n/2**32) + n.
shrsi q,q,2          q = floor(q/4).
shri  t,n,31         Add 1 to q if
add   q,q,t          n is negative.

muli  t,q,7          Compute remainder from
sub   r,n,t          r = n - q*7.
```

证明 重要的是要记住，上面代码中的指令“add q, q, n”不产生溢出。这是因为由于乘的是一个负数，所以q和n的符号相反。这样，“计算机算数”加法与实数加法相同。因此，对于$n>0$，上面的代码计算

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \times \frac{n}{2^{32}} \right\rfloor + n \right) / 4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n + 5n - 7\times 2^{32}n + 7\times 2^{32}n}{7\times 2^{32}} \right\rfloor / 4 \right\rfloor$$

$$= \left\lfloor \frac{n}{7} + \frac{5n}{7\times 2^{34}} \right\rfloor$$

这里，我们使用了定理D3的推论。

对于$0 \leqslant n < 2^{31}$，误差项$5n/7 \times 2^{34}$是非负的，而且小于1/7，所以$q = \lfloor n/7 \rfloor$。

对于$n<0$，上面的代码计算

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \times \frac{n}{2^{32}} \right\rfloor + n \right) / 4 \right\rfloor + 1 = \left\lceil \frac{n}{7} + \frac{5n+1}{7\times 2^{34}} \right\rceil$$

误差项是非正的，而且大于$-1/7$，所以$q = \lceil n/7 \rceil$。

用1个移位量为3的左移位和1个减可以实现立即乘。

10.4 除数>2的带符号除法

这里，读者可能要问，其他除数是否会有其他问题，在本节我们会看到，答案是不会；前面给出的三个例子展示了（$d>2$时）仅有的几种出现问题的情况。

有些证明部分有点复杂，所以要小心。我们对字长W进行证明。 160

给定字长$W>3$和除数d，$2<d<2^{W-1}$，我们希望寻找最小整数m和整数p，使得$0<m<2^W$、$p>W$，而且满足：

$$\text{对于 } 0 \leqslant n < 2^{W-1},\ \left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \tag{10-1a}$$

$$\text{对于 } -2^{W-1} \leqslant n \leqslant -1,\ \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \tag{10-1b}$$

需要最小整数m的原因是，更小的乘数可以给出更小的移位量（可能是零）或生成类似于“除以5”例子而不是“除以7”例子中的代码。必须要求$m<2^W-1$，这样代码就不需要比“除以7”例子中的代码处理更多的指令。（也就是说，虽然可以使用在“除以7”例子中所用的插入add的方法来处理范围在2^{W-1}到2^W-1的乘数，但是我们不愿意处理更大的乘数。）要求有$p>W$，因为生成的代码提取积mn的左半部分，这等价于把mn向右移位W个位置。因此总的右

移位量是W或更多。

乘数m和用M标记的“魔术数”是不同的。魔术数是用于乘指令的值，它由下式给出：

$$M = \begin{cases} m, & \text{如果} 0 \leqslant m < 2^{W-1} \\ m - 2^W, & \text{如果} 2^{W-1} \leqslant m < 2^W \end{cases}$$

因为式(10-1b)对于$n=-d$一定成立，所以$\lfloor -md/2^p \rfloor + 1 = -1$，这表明

$$\frac{md}{2^p} > 1 \tag{10-2}$$

设n_c是使$\text{rem}(n_c, d)=d-1$成立的n的最大（正）值。这样的n_c一定存在，因为$n_c=d-1$满足$\text{rem}(n_c, d)=d-1$。可以通过计算$n_c = \lfloor 2^{W-1}/d \rfloor d - 1 = 2^{W-1} - \text{rem}(2^{W-1}, d) - 1$来求这个$n_c$。$n_c$是$n$的最大$d$可接受值，所以有

$$2^{W-1} - d \leqslant n_c \leqslant 2^{W-1} - 1 \tag{10-3a}$$

而且显然有

161

$$n_c \geqslant d - 1 \tag{10-3b}$$

因为式(10-1a)对于$n=n_c$一定成立，所以有

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

或

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}$$

把这一表达式与(10-2)结合，得出：

$$\frac{2^p}{d} < m < \frac{2^p}{d} \times \frac{n_c + 1}{n_c} \tag{10-4}$$

因为m是满足式(10-4)的最小整数，它是比$2^p/d$大的下一个整数；即，

$$\boxed{m = \frac{2^p + d - \text{rem}(2^p, d)}{d}} \tag{10-5}$$

把这个表达式与式(10-4)的右边结合，经简化后得到：

$$\boxed{2^p > n_c(d - \text{rem}(2^p, d))} \tag{10-6}$$

1. 算法

由上可知，寻找魔术数M和根据除数d求移位量s的算法首先计算n_c，然后再通过尝试下一个更大值的方式求满足式(10-6)的p。如果$p<W$，就设$p=W$（由下面的定理，这个p也满足式(10-6)）。当找到满足式(10-6)的最小的$p \geqslant W$时，可以通过计算式(10-5)得到m。这就是m的最小可能值，因为找到的是最小可接受的p，那么根据式(10-4)，显然p的更小值产生更小的m值。最后，$s=p-W$而且M是m的带符号整数解释（这就是`mulhs`指令对它的解释）。

下面的定理证明可以令p最小为W：

定理DC1 如果式(10-6)对某个p为真，那么它对所有更大的p也为真。

证明 假设(10-6)对$p=p_0$为真，把式(10-6)乘以2得到:

$$2^{p_0+1} > n_c(2d - 2\text{rem}(2^{p_0}, d))$$

162

根据定理D5，$\text{rem}(2^{p_0+1}, d) \geqslant 2\text{rem}(2^{p_0}, d)-d$。与上式结合可以得到

$$2^{p_0+1} > n_c(2d - (\text{rem}(2^{p_0+1}, d) + d)) \quad \text{或}$$

$$2^{p_0+1} > n_c(d - \text{rem}(2^{p_0+1}, d))$$

因此，式(10-6)对$p=p_0+1$为真，因此对所有更大的值为真。

因此，可以通过二分查找来解式(10-6)，虽然，简单的线性查找可能更好（从$p=W$开始），因为d通常很小，而小的d值导致小的p值。

2. 证明这一算法的可行性

必须证明式(10-6)总有解，而且$0 < m < 2^W$。(没有必要证明$p \geqslant W$，因为这是强制的。)

通过得到p的上界来证明式(10-6)总有解。为全局考虑，我们还推导出在假设p的最小值不强制为W的情况下的p的下界。为了得到p的上下界，注意，对任意的正整数x，总存在大于x而小于或等于$2x$的2的幂。因此由(10-6)，有

$$n_c(d - \text{rem}(2^p, d)) < 2^p \leqslant 2n_c(d - \text{rem}(2^p, d))$$

因为$0 \leqslant \text{rem}(2^p, d) \leqslant d-1$，有

$$n_c + 1 \leqslant 2^p \leqslant 2n_c d \tag{10-7}$$

由式(10-3a)和(10-3b)，$n_c \geqslant \max(2^{W-1}-d, d-1)$。直线$f_1(d)=2^{W-1}-d$与直线$f_2(d)=d-1$在$d=(2^{W-1}+1)/2$处相交。因此，$n_c \geqslant (2^{W-1}-1)/2$。因为$n_c$是一个整数，有$n_c \geqslant 2^{W-2}$。由此及$d < 2^{W-1}-1$，所以式(10-7)变成

$$2^{W-2} + 1 \leqslant 2^p \leqslant 2(2^{W-1} - 1)^2$$

或

$$W - 1 \leqslant p \leqslant 2W - 2 \tag{10-8}$$

(例如，当$W=32, d=3$时）能够达到下界$p=W-1$，但是在这时，我们设置$p=W$。

如果不强制令p等于W，那么由式(10-4)和(10-7)，有

$$\frac{n_c + 1}{d} < m < \frac{2n_c d}{d} \times \frac{n_c + 1}{n_c}$$

163

由式(10-3b)，得

$$\frac{d - 1 + 1}{d} < m < 2(n_c + 1)$$

因为$n_c < 2^{W-1}-1$（由式10-3a)，有

$$2 \leqslant m \leqslant 2^W - 1$$

如果令$p=W$，由式(10-4)，有

$$\frac{2^W}{d} < m < \frac{2^W}{d} \times \frac{n_c + 1}{n_c}$$

因为$2 < d < 2^{W-1} - 1$, $n_c > 2^{W-2}$，有

$$\frac{2^W}{2^{W-1} - 1} < m < \frac{2^W}{2} \times \frac{2^{W-2} + 1}{2^{W-2}}$$，或

$$3 \leqslant m \leqslant 2^{W-1} + 1$$

因此，在任何情况之下，m都在“除以7”的例子所示的代码的限制之内。

3. 证明积是正确的

必须证明如果p和m是由式(10-6)和(10-5)计算而来，那么等式(10-1a)和(10-1b)成立。

容易看出，等式(10-5)和不等式(10-6)蕴含着式(10-4)。（在强制令p等于W的情况下，如定理DC1所示，不等式(10-6)仍然成立。）下面我们分别考虑n的以下五种取值范围：

$$0 \leqslant n \leqslant n_c,$$

$$n_c + 1 \leqslant n \leqslant n_c + d - 1,$$

$$-n_c \leqslant n \leqslant -1,$$

$$-n_c - d + 1 \leqslant n \leqslant -n_c - 1, \quad 及$$

$$n = -n_c - d$$

由式(10-4)，因为m是整数，有

$$\frac{2^p}{d} < m \leqslant \frac{2^p(n_c + 1) - 1}{dn_c}$$

164

将不等式两边乘以$n/2^p$，对于$n > 0$，上面的不等式变成：

$$\frac{n}{d} \leqslant \frac{mn}{2^p} \leqslant \frac{2^p n(n_c + 1) - n}{2^p dn_c}$$，所以

$$\left\lfloor \frac{n}{d} \right\rfloor \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor \leqslant \left\lfloor \frac{n}{d} + \frac{(2^p - 1)n}{2^p dn_c} \right\rfloor$$

因为$0 < n < n_c$，$0 < (2^p - 1)n/(2^p dn_c) < 1/d$，所以根据定理D4，有

$$\left\lfloor \frac{n}{d} + \frac{(2^p - 1)n}{2^p dn_c} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor$$

因此，在这种情况下（$0 < n < n_c$），式(10-1a)成立。

对于$n > n_c$，n被限制于下面的范围

$$n_c + 1 \leqslant n \leqslant n_c + d - 1 \tag{10-9}$$

这是因为$n > n_c + d$与n_c为使$\mathrm{rem}(n_c, d) = d - 1$成立的$n$的最大值的选择相矛盾（换言之，由式(10-3a)，$n > n_c + d$导致$n > 2^{W-1}$）。由式(10-4)，对于$n > 0$，有

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n}{d} \times \frac{n_c + 1}{n_c}$$

根据初等代数知识，上面的表达式可以写成：

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n_c+1}{d} + \frac{(n-n_c)(n_c+1)}{dn_c} \tag{10-10}$$

由式(10-9)，$1 \leqslant n-n_c \leqslant d-1$，所以有

$$0 < \frac{(n-n_c)(n_c+1)}{dn_c} \leqslant \frac{d-1}{d} \times \frac{n_c+1}{n_c}$$

因为$n_c \geqslant d-1$（根据式(10-3b)）以及当n_c取最小值时$(n_c+1)/n_c$取最大值，有

$$0 < \frac{(n-n_c)(n_c+1)}{dn_c} \leqslant \frac{d-1}{d} \times \frac{d-1+1}{d-1} = 1$$

165

在式(10-10)中，项$(n_c+1)/d$是整数。项$(n-n_c)(n_c+1)/dn_c$小于或等于1。因此，式(10-10)变成

$$\left\lfloor \frac{n}{d} \right\rfloor \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor \leqslant \frac{n_c+1}{d}$$

对于范围在式(10-9)中的所有n，$\lfloor n/d \rfloor = (n_c+1)/d$。因此，在这种情况下（$n_c+1 \leqslant n \leqslant n_c+d-1$）式(10-1a)成立。

对于$n<0$，因为m是整数，由式(10-4)，有

$$\frac{2^p+1}{d} \leqslant m < \frac{2^p}{d} \times \frac{n_c+1}{n_c}$$

对上面的表达式的两边同时乘以$n/2^p$，对于$n<0$，有

$$\frac{n}{d} \times \frac{n_c+1}{n_c} < \frac{mn}{2^p} \leqslant \frac{n}{d} \times \frac{2^p+1}{2^p}$$

或

$$\left\lfloor \frac{n}{d} \times \frac{n_c+1}{n_c} \right\rfloor + 1 \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \leqslant \left\lfloor \frac{n}{d} \times \frac{2^p+1}{2^p} \right\rfloor + 1$$

利用定理D2可得

$$\left\lceil \frac{n(n_{c+1}) - dn_c + 1}{dn_c} \right\rceil + 1 \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \leqslant \left\lceil \frac{n(2^p+1) - 2^p d + 1}{2^p d} \right\rceil + 1,$$

$$\left\lceil \frac{n(n_c+1)+1}{dn_c} \right\rceil \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \leqslant \left\lceil \frac{n(2^p+1)+1}{2^p d} \right\rceil$$

因为$n+1 \leqslant 0$，可以放宽右边的不等式，得出

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \leqslant \left\lceil \frac{n}{d} \right\rceil \tag{10-11}$$

对于$-n_c \leqslant n \leqslant -1$，有

$$\frac{-n_c+1}{dn_c} \leqslant \frac{n+1}{dn_c} \leqslant 0, \text{ 或}$$

$$-\frac{1}{d} < \frac{n+1}{dn_c} \leqslant 0$$

166

因此，根据定理D4，有

$$\left\lceil \frac{n}{d} + \frac{n+1}{dn_c} \right\rceil = \left\lceil \frac{n}{d} \right\rceil$$

所以在这种情况下（$-n_c < n < -1$），式(10-1b)成立。

对于$n<-n_c$，n局限于下面的范围

$$-n_c - d \leqslant n \leqslant -n_c - 1 \tag{10-12}$$

（根据式(10-3a)，$n<-n_c-d$可以推出$n<-2^{W-1}$，这是不可能的）。对不等式（10-11）的左边做初等代数变换，有

$$\left\lceil \frac{-n_c-1}{d} + \frac{(n+n_c)(n_c+1)+1}{dn_c} \right\rceil \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \leqslant \left\lceil \frac{n}{d} \right\rceil \tag{10-13}$$

对于$-n_c-d < n < -n_c-1$，有

$$\frac{(-d+1)(n_c+1)}{dn_c} + \frac{1}{dn_c} \leqslant \frac{(n+n_c)(n_c+1)+1}{dn_c} \leqslant \frac{-(n_c+1)+1}{dn_c} = -\frac{1}{d}$$

当n_c取最小值时，比值$(n_c+1)/n_c$取最大值；也就是说，$n_c=d-1$。因此，

$$\frac{(-d+1)(d-1+1)}{d(d-1)} + \frac{1}{dn_c} \leqslant \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0, \text{ 或}$$

$$-1 < \frac{(n+n_c)(n_c+1)+1}{dn_c} < 0$$

因为$(-n_c-1)/d$是整数，而且在式（10-13）中与它相加的量在0与−1之间，所以有

$$\frac{-n_c-1}{d} \leqslant \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \leqslant \left\lceil \frac{n}{d} \right\rceil$$

因为n满足$-n_c-d+1 < n < -n_c-1$，有

$$\left\lceil \frac{n}{d} \right\rceil = \frac{-n_c-1}{d}$$

因此$\lfloor mn/2^p \rfloor + 1 = \lceil n/d \rceil$，即式(10-1b)成立。

最后的情况，$n=-n_c-d$，只能在d的特定值下发生。由式(10-3a)，有$-n_c-d < -2^{W-1}$，所以，如果n取这个值，一定要有$n=-n_c-d=-2^{W-1}$，因此，$n_c=2^{W-1}-d$。因此，$\text{rem}(2^{W-1}, d) = \text{rem}(n_c+d, d) = d-1$（也就是说，$d$整除$2^{W-1}+1$）。

167

对于这种情况（$n=-n_c-d$），式(10-6)有解$p=W-1$（p的最小可能值），因为对$p=W-1$，有

$$\begin{aligned} n_c(d - \text{rem}(2^p, d)) &= (2^{W-1} - d)(d - \text{rem}(2^{W-1}, d)) \\ &= (2^{W-1} - d)(d - (d-1)) = 2^{W-1} - d < 2^{W-1} = 2^p \end{aligned}$$

那么根据式(10-5)，有

$$m = \frac{2^{W-1} + d - \text{rem}(2^{W-1}, d)}{d} = \frac{2^{W-1} + d - (d-1)}{d} = \frac{2^{W-1} + 1}{d}$$

所以，有

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lfloor \frac{2^{W-1}+1}{d} \times \frac{-2^{W-1}}{2^{W-1}} \right\rfloor + 1 = \left\lfloor \frac{-2^{W-1}-1}{d} \right\rfloor + 1$$
$$= \left\lceil \frac{-2^{W-1}-d}{d} \right\rceil + 1 = \left\lceil \frac{-2^{W-1}}{d} \right\rceil = \left\lceil \frac{n}{d} \right\rceil$$

所以式(10-1b)成立。

这就完成了证明，即如果m和p是由式(10-5)和(10-6)计算而来的，那么等式(10-1a)和(10-1b)对所有n的可接受值都成立。

10.5 除数 < -2的带符号除法

因为带符号整数除法满足$n \div (-d) = -(n \div d)$，只要生成计算$n \div |d|$的代码，再在其后面加一个对商求负的指令即可。（当$d = -2^{W-1}$不能产生正确的结果，但是对于这一除数和其他负的2的幂的除数，可以使用10.1节的代码，然后再加一个取负的指令。）因为有取最大负数的可能，不对被除数取负。

然而，回避取负指令是可能的，办法是计算如下表达式：

$$\text{如果 } n \leqslant 0,\ q = \left\lfloor \frac{mn}{2^p} \right\rfloor$$

$$\text{如果 } n > 0,\ q = \left\lfloor \frac{mn}{2^p} \right\rfloor + 1$$

但是，如果$n>0$，则加1的实现很困难（因为不能简单地使用n的符号位），所以代码将代之以如果$q<0$则加1。因为乘数m是负的，所以这是等价的。 168

$W=32$，$d=-7$时的代码如下所示。

```
li    M,0x6DB6DB6D  Magic num, -(2**34+5)/7 + 2**32.
mulhs q,M,n         q = floor(M*n/2**32).
sub   q,q,n         q = floor(M*n/2**32) - n.
shrsi q,q,2         q = floor(q/4).
shri  t,q,31        Add 1 to q if
add   q,q,t         q is negative (n is positive).

muli  t,q,-7        Compute remainder from
sub   r,n,t         r = n - q*(-7).
```

这一代码与除数是+7时的代码相同，只是它使用了+7时的乘数的负值，而且在乘法后面使用的不是`add`而是`sub`，另外，如上讨论的那样，31位的`shri`必须是对`q`而不是对n进行的。（对于$d=+7$的情况也可使用对`q`的移位操作，但会降低代码的并行性。）减指令不产生溢出，因为操作数的符号相同。然而，这一方法不是总能奏效！尽管上面的代码对$W=32$、$d=-7$是正确的，将“除以3”的代码类似地修改成除以-3的代码时，对$W=32$、$n=-2^{31}$就不能给出正确的结果。

我们对这一情形做更进一步的观察。

给定一个字长$W>3$和一个除数d，$-2^{W-1}<d<-2$，我们希望找到一个（绝对值）最小的整数m和p，使得

$$\text{对于} -2^{W-1} \leqslant n \leqslant 0,\ \left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \tag{10-14a}$$

$$\text{对于} 1 \leqslant n < 2^{W-1},\ \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \tag{10-14b}$$

对$-2^W < m < 0$和$p \geqslant W$成立。

与处理正除数的除法相类似，设n_c是满足以下条件的n的最大负值：存在整数k，使得$n_c=kd+1$。这样的n_c存在，因为$n_c=d+1$是一个可能值。可以从$n_c=\lfloor(-2^{W-1}-1)/d\rfloor\, d+1=-2^{W-1}+\text{rem}(2^{W-1}+1, d)$计算它。$n_c$是$n$的一个最小$|d|$可接受值，所以有

$$-2^{W-1} \leqslant n_c \leqslant -2^{W-1} - d - 1 \tag{10-15a}$$

而且，显然有

169

$$n_c \leqslant d + 1 \tag{10-15b}$$

因为式(10-14b)对$n=-d$一定成立，而且式(10-14a)对$n=n_c$一定成立，所以得到如下类似于式(10-4)的不等式

$$\frac{2^p}{d} \times \frac{n_c - 1}{n_c} < m < \frac{2^p}{d} \tag{10-16}$$

因为m是满足式(10-16)的最大整数，这是小于$2^p/d$的下一个整数，也就是说

$$\boxed{m = \frac{2^p - d - \text{rem}(2^p, d)}{d}} \tag{10-17}$$

将这个表达式与式(10-16)的左侧结合并简化，得出

$$\boxed{2^p > n_c(d + \text{rem}(2^p, d))} \tag{10-18}$$

式(10-17)和(10-18)所表明的算法的可行性证明以及积的正确性证明与正除数的情况类似，在此省略。然而，$-2^W < m < 0$的证明会有一些困难。为了证明这个不等式，分别考虑下面的情况：d是2的幂的负数或其他数。当$d=-2^k$时，很容易证明$n_c=-2^{W-1}+1$、$p=W+k-1$和$m=-2^{W-1}-1$（它在要求的范围之内）。对于d不是-2^k形式的情况，将前面的证明修改一下就可以立即得到证明。

除数在什么情况下有$m(-d) \neq -m(d)$

我们用$m(d)$表示对应于除数d的乘数。如果$m(-d)=-m(d)$，那么，可以通过计算$|d|$的乘数，再取它的负值，然后生成与上面所讲述的“除以-7”类似的代码，从而得到除以$-d$的代码。

通过把式(10-18)与式(10-6)、式(10-17)与式(10-5)比较，可以看出，如果对应于$-d$的n_c是对应于d的n_c的值的负值时，那么就有$m(-d)=-m(d)$。因此，只有当对应于负除数的n_c值是最大负数-2^{W-1}时，才有$m(-d) \neq -m(d)$。这样的除数是$2^{W-1}+1$的负因子。这些数相当少见，如下所示，可以通过因数分解得到：

$$2^{15}+1 = 3\times 11\times 331$$
$$2^{31}+1 = 3\times 715\ 827\ 883$$
$$2^{63}+1 = 3^3\times 19\times 43\times 5419\times 77\ 158\ 673\ 929$$

170

对于所有这些因子，$m(-d)\neq -m(d)$。证明概述如下：对于$d>0$，我们有$n_c=2^{W-1}-d$。因为$\mathrm{rem}(2^{W-1}, d)=d-1$，$p=W-1$满足式(10-6)，因此$p=W$也满足式(10-6)。然而，对于$d<0$，我们有$n_c=-2^{W-1}$和$\mathrm{rem}(2^{W-1}, d)=|d|-1$。因此，对于$p=W-1$或$p=W$，式(10-18)不成立，所以$p>W$。

10.6 并入编译器

为了使编译器把除以常量的除法变成乘法，给定一个除数d，它必须计算对应于d的魔术数M和移位量s。简洁的计算就是对$p=W, W+1\cdots$计算式(10-6)或(10-18)，直到其成立。然后，可以用式(10-5)或(10-17)来计算m。这时，M就是m的带符号整数表示，而且$s=p-W$。

下面所描述的设计用一小段额外的代码处理正的和负的d，而且这一设计不使用双字算术。

回想一下，n_c是由下式给出的：

$$n_c = \begin{cases} 2^{W-1}+\mathrm{rem}(2^{W-1}, d)-1, & \text{如果 } d>0 \\ -2^{W-1}+\mathrm{rem}(2^{W-1}+1, d), & \text{如果 } d<0 \end{cases}$$

因此，可以通过下式计算$|n_c|$：

$$t = 2^{W-1}+\begin{cases} 0, & \text{如果 } d>0 \\ 1, & \text{如果 } d<0 \end{cases}$$
$$|n_c| = t-1-\mathrm{rem}(t, |d|)$$

鉴于参数的取值，必须使用无符号除法来计算余数。上面我们写作$\mathrm{rem}(t,|d|)$而不是$\mathrm{rem}(t,d)$，是为了强调算法必须处理的是两个正（而且是无符号的）参数。

由式(10-6)和(10-18)，可以从下面的式子求得p

$$2^p > |n_c|(|d|-\mathrm{rem}(2^p, |d|)) \tag{10-19}$$

然后从式(10-5)和(10-17)可以求得$|m|$：

$$|m| = \frac{2^p+|d|-\mathrm{rem}(2^p, |d|)}{|d|} \tag{10-20}$$

直接计算式(10-19)中的$\mathrm{rem}(2^p, |d|)$需要“长除法”（$2W$位的被除数除以W位的除数，得到W位的商和余数），而且实际上这个除法必须是无符号长除法。有一种解式(10-19)的方法，这一方法实际上完成所有的计算，它回避了长除法，而且很容易在传统HLL中仅用W位算术来实现。然而，我们的确需要无符号除法和无符号比较。

171

我们可以递增地计算$\mathrm{rem}(2^p, |d|)$，首先，对于$p=2^{W-1}$，初始化两个变量q和r为$2^p/|d|$的商和余数，然后再递增p，并同时更新r和q。

随着寻找过程的进行，即当对p递增1时，q和r由下面的代码（参见定理D5(a)）来更新。

```
q = 2*q;
r = 2*r;
if (r >= abs(d)) {
   q = q + 1;
   r = r - abs(d);}
```

由不等式(10-4)的左边、(10-16)的右边以及给出的m的界，可以得到$q=\lfloor 2^p/|d| \rfloor < 2^W$，所以$q$可以表示成$W$位无符号整数。同样，因为$0 \le r < |d|$，所以，$r$也可表示成$W$位带符号或无符号整数。(警告：中间结果$2r$可能会超过$2^{W-1}-1$，所以$r$应该是无符号的，而且上面的比较也应该是无符号比较。)

下一步计算$\delta=|d|-r$。这一减法中的两个项都可表示成W位无符号整数，结果也是如此($1 \le \delta \le |d|$)，所以这里没有什么困难。

为了回避式(10-19)的长乘法，我们把它改写成：

$$\frac{2^p}{|d|} > \delta$$

量$2^p/|n_c|$可以表示成W位无符号整数(类似于式(10-7)，由式(10-19)可以证明$2^p \le 2/|n_c| \cdot |d|$，

```
struct ms {int M;              // Magic number
           int s;};            // and shift amount.

struct ms magic(int d) {       // Must have 2 <= d <= 2**31-1
                               // or   -2**31 <= d <= -2.
   int p;
   unsigned ad, anc, delta, q1, r1, q2, r2, t;
   const unsigned two31 = 0x80000000;     // 2**31.
   struct ms mag;

   ad = abs(d);
   t = two31 + ((unsigned)d >> 31);
   anc = t - 1 - t%ad;         // Absolute value of nc.
   p = 31;                     // Init. p.
   q1 = two31/anc;             // Init. q1 = 2**p/|nc|.
   r1 = two31 - q1*anc;        // Init. r1 = rem(2**p, |nc|).
   q2 = two31/ad;              // Init. q2 = 2**p/|d|.
   r2 = two31 - q2*ad;         // Init. r2 = rem(2**p, |d|).
   do {
      p = p + 1;
      q1 = 2*q1;               // Update q1 = 2**p/|nc|.
      r1 = 2*r1;               // Update r1 = rem(2**p, |nc|.
      if (r1 >= anc) {         // (Must be an unsigned
         q1 = q1 + 1;          // comparison here).
         r1 = r1 - anc;}
      q2 = 2*q2;               // Update q2 = 2**p/|d|.
      r2 = 2*r2;               // Update r2 = rem(2**p, |d|.
      if (r2 >= ad) {          // (Must be an unsigned
         q2 = q2 + 1;          // comparison here).
         r2 = r2 - ad;}
      delta = ad - r2;
   } while (q1 < delta || (q1 == delta && r1 == 0));

   mag.M = q2 + 1;
   if (d < 0) mag.M = -mag.M; // Magic number and
   mag.s = p - 32;            // shift amount to return.
   return mag;
}
```

图10-1 带符号除法的魔术数计算

而且，因为$d=-2^{W-1}$、$n_c=-2^{W-1}+1$和$p=2W-2$，所以对于$W>3$有$2^p/|n_c|=2^{2W-2}/(2^{W-1}-1)<2^W$)。同样，可以用与rem($2^p$, $|d|$)的递增计算相同的方法递增地（递增p）计算这个量$2^p/|n_c|$。由于$2^p/|n_c|>2^{W-1}$的情况（当d很大时，这种情况就会发生），这里的比较也应该是无符号的。

要计算m，不需要直接计算式(10-20)（计算式(10-20)需要长除法）。我们有

$$\frac{2^p+|d|-\text{rem}(2^p,|d|)}{|d|}=\left\lfloor\frac{2^p}{|d|}\right\rfloor+1=q+1$$

循环结束检测$2^p/|n_c|>\delta$很难计算。只有通过商q_1和余数r_1才能得到$2^p/|n_c|$。$2^p/|n_c|$可能是整数，也可能不是整数（只有当$d=2^{W-2}+1$及d取某些负值时，它才是整数）。检测$2^p/|n_c|\leqslant\delta$可以编码如下： 172

$$q_1<\delta\mid(q_1=\delta\ \&\ r_1=0)$$

从d计算M和s的完整过程如图10-1所示，这一过程使用C语言，适用于$W=32$的情况。有几个地方可能产生溢出，但是如果忽略溢出的话，可以得到正确的结果。

要使用这个程序的结果，编译器应该生成li和mulhs指令，如果d>0、M<0则生成add指令，如果d<0、M>0则生成sub指令，如果s>0则生成shrsi指令。然后，一定要生成shri指令和最后的add指令。

对于$W=32$，通过下面的处理可以回避处理负除数的情况：对于$d=3$和$d=715\ 827\ 883$返回预先计算的相应结果；对其他负除数，运用$m(-d)=-m(d)$。然而，这样的算法即便比图10-1所示的算法短也不会短太多。

10.7 其他主题

定理DC2 *如果不强令p等于W的话，那么最小的乘数m是奇数。*

证明 假设对于最小整数p和偶数m，等式(10-1a)和(10-1b)成立。那么显然，对$m/2$和$p-1$，等式(10-1a)和(10-1b)仍然成立。这与p最小的假设矛盾。

1. 惟一性

对于给定的除数，有时魔术数是惟一的（例如，$W=32$、$d=7$时），但是通常它不是惟一的。事实上，经验表明魔术数通常不是惟一的。例如，对于$W=32$、$d=6$，有四个魔术数：

$$\begin{aligned}
M &= 715\,827\,833 \quad ((2^{32}+2)/6), && s=0\\
M &= 1\,431\,655\,766 \quad ((2^{32}+2)/3), && s=1\\
M &= -1\,431\,655\,765 \quad ((2^{33}+1)/3-2^{32}), && s=2\\
M &= -1\,431\,655\,764 \quad ((2^{33}+4)/3-2^{32}), && s=2
\end{aligned}$$

然而，它们有下面的惟一性性质：

定理DC3 *对于给定的除数d，如果不强令p等于W，那么有p的最小值的乘数m是惟一的。* 173

证明 首先考虑$d>0$的情况。不等式(10-4)的上限与下限的差是$2^p/dn_c$。我们已经证明，如果p是最小的，那么有$2^p/dn_c<2$（见式10-7）。因此，最多有两个满足(10-4)的m。设m是由式(10-5)给出的这些值中较小的那个值；那么$m+1$就是另外一个值。 174

对于乘数$m+1$，设p_0是使式(10-4)的右边成立的p的最小值（不强令p_0等于W）。那么有

$$\frac{2^{p_0}+d-\text{rem}(2^{p_0},d)}{d}+1<\frac{2^{p_0}}{d}\times\frac{n_c+1}{n_c}$$

简化得到

$$2^{p_0}>n_c(2d-\text{rem}(2^{p_0},d))$$

两边除以2，得

$$2^{p_0-1}>n_c\left(d-\frac{1}{2}\text{rem}(2^{p_0},d)\right)$$

因为$\text{rem}(2^{p_0},d)<2\text{rem}(2^{p_0-1},d)$（由9.1节中介绍的定理D5可知），有

$$2^{p_0-1}>n_c(d-\text{rem}(2^{p_0-1},d))$$

与p_0是最小值的假设矛盾。

对于$d<0$的证明类似，在此不再给出。

2. 具有最优代码的除数

对于$d=3$、$W=32$，程序代码特别短，因为在`mulhs`指令之后没有`add`和`shrsi`指令。具有这样短代码的其他除数是什么呢?

我们只考虑正除数的情况。我们希望找到整数m和p，使得它们满足等式(10-1a)和(10-1b)，而且$p=W$，$0<m<2^{W-1}$。因为任意满足等式(10-1a)和(10-1b)的整数m和p也一定满足式(10-4)，所以只要找到这样的除数d就可以了：对于它，式(10-4)有一个$p=W$和$0<m<2^{W-1}$的解。式(10-4)的所有$p=W$的解由下面的式子给出:

$$m=\frac{2^W+kd-\text{rem}(2^W,d)}{d},\quad k=1,2,3,\ldots.$$

把上式与式(10-4)的右边结合化简，得

$$\text{rem}(2^W,d)>kd-\frac{2^W}{n_c} \tag{10-21}$$

对$\text{rem}(2^W,d)$的最弱限制是$k=1$及n_c为最小的2^{W-2}。因此一定有:

$$\text{rem}(2^W,d)>d-4$$

175 也就是说，d整除2^W+1、2^W+2或2^W+3。

现在让我们看一看这些因子中哪一个有最优代码。

如果d整除2^W+1，那么$\text{rem}(2^W,d)=d-1$。于是，式(10-6)的一个解是$p=W$，这是因为不等式(10-6)变成

$$2^W>n_c(d-(d-1))=n_c$$

因为$n_c<2^{W-1}$，这一不等式显然成立。那么，计算m可得

$$m=\frac{2^W+d-(d-1)}{d}=\frac{2^W+1}{d}$$

对于$d>3$（因为d整除2^W+1，$d\neq 2$），m显然小于2^{W-1}。因此，2^W+1的所有因子都有最优

代码。

类似地，如果d整除2^W+2，那么rem$(2^W, d)=d-2$。同样，式(10-6)的一个解是$p=W$，这是因为不等式(10-6)变成

$$2^W > n_c(d-(d-2)) = 2n_c$$

这显然为真。那么，计算m可得

$$m = \frac{2^W + d - (d-2)}{d} = \frac{2^W + 2}{d}$$

对于$d=2$，这个m大于$2^{W-1}-1$，但是对于$W \geq 3$及$d \geq 3$，它小于或等于$2^{W-1}-1$（$W=3$、$d=3$的情况不会发生，因为3不是$2^3+2=10$的因子）。因此，除了2和2的余因子之外，2^W+2的所有因子都有最优代码。（2的余因子是$(2^W+2)/2$，它们不能表示成W位的带符号整数。）

如果d整除2^W+3，下面的论证表明d没有最优代码。因为rem$(2^W, d)=d-3$，由不等式(10-21)可知，存在$k=1,2,3,...$使得下式成立：

$$n_c < \frac{2^W}{kd - d + 3}$$

这一不等式的最弱限制是$k=1$，所以我们一定有$n_c<2^W/3$。

由式(10-3a)可知，$n_c \geq 2^{W-1}-d$或$d \geq 2^{W-1}-n_c$。因此有

$$d > 2^{W-1} - \frac{2^W}{3} = \frac{2^W}{6}$$

176

同样，因为2、3和4不能整除2^W+3，2^W+3的最小可能因子是5。因此，最大的可能因子是$(2^W+3)/5$。所以，如果d整除2^W+3，而且d有最优代码，那么一定有

$$\frac{2^W}{6} < d \leq \frac{2^W + 3}{5}$$

对这个不等式的各项取对应于2^W+3的倒数，即，取倒数后再乘以2^W+3。由此，d的余因子$(2^W+3)/d$必须满足：

$$5 \leq \frac{2^W + 3}{d} < \frac{(2^W + 3) \times 6}{2^W} = 6 + \frac{18}{2^W}$$

对于$W \geq 5$，这一不等式显然表明可能的余因子只有5和6。对于$W<5$，很容易证实2^W+3没有因子。因为6不能是2^W+3的因子，惟一可能的因子是5。因此，2^W+3的有最优代码的惟一可能因子是$(2^W+3)/5$。

对于$d=(2^W+3)/5$，有

$$n_c = \left\lfloor \frac{2^{W-1}}{(2^W+3)/5} \right\rfloor \left(\frac{2^W+3}{5} \right) - 1$$

对于$W \geq 4$，有

$$2 < \frac{2^{W-1}}{(2^W+3)/5} < 2.5$$

所以有

$$n_c = 2 \times \left(\frac{2^W + 3}{5}\right) - 1$$

这个n_c值大于$2^W/3$，所以$d=(2^W+3)/5$不能有最优代码。因为对于$W<4$，2^W+3不存在因子，所以我们得出2^W+3没有最优代码的结论。

总之，2^W+1和2^W+2的所有因子，除2和$(2^W+2)/2$之外，都有最优代码，其他的数没有最优代码。另外，以上的证明表明，当一个数存在最优代码时，（图10-1的）算法*magic*总生成最优代码。

177

让我们考虑$W=16$、32和64的特殊情况。相关的因数分解如下所示：

$$2^{16}+1 = 65537 \text{（素数）} \qquad 2^{32}+1 = 641 \times 6\,700\,417$$
$$2^{16}+2 = 2\times 3^2 \times 11 \times 331 \qquad 2^{32}+2 = 2\times 3\times 715\,827\,883$$
$$2^{64}+1 = 274\,177 \times 67\,280\,421\,310\,721$$
$$2^{64}+2 = 2\times 3^3 \times 19 \times 43 \times 5419 \times 77\,158\,673\,929$$

因此，对于$W=16$，我们得到有20个除数有最优代码的结果。其中，小于100的除数有3、6、9、11、18、22、33、66和99。

对于$W=32$，有6个这样的除数：3、6、641、6 700 417、715 827 883和1 431 655 766。

对于$W=64$，有126个这样的除数，小于100的这样的除数是：3、6、9、18、19、27、38、43、54、57和86。

10.8 无符号除法

当然除以2的幂的无符号除法可以通过一个*逻辑右移位*实现，而余数可以通过*立即与*实现。

似乎对于其他除数的处理会很简单：只要使用$d>0$的带符号除法的结果，省略两个如果商为负则加1的指令即可。然而我们将看到，在这样的无符号除法的情况下，某些细节实际上很复杂。

1. 除以3的无符号除法

对于非2的幂的数，让我们首先考虑在32位计算机上的除以3的无符号除法。因为被除数n最大可以是$2^{32}-1$，乘数$(2^{32}+2)/3$不合适，因为误差项$2n/3 \times 2^{32}$（参看上面除以3的例子）可能大于1/3。然而，乘数$(2^{33}+1)/3$是合适的。代码是：

```
li    M,0xAAAAAAAB  Load magic number, (2**33+1)/3.
mulhu q,M,n         q = floor(M*n/2**32).
shri  q,q,1

muli  t,q,3         Compute remainder from
sub   r,n,t         r = n - q*3.
```

需要1个给出64位积的高阶32位的指令，在上面的代码中这一指令被记为mulhu。

为了确认这一代码的正确性，注意，它计算：

178

$$q = \left\lfloor \frac{2^{33}+1}{3} \times \frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{n}{3\times 2^{33}} \right\rfloor$$

对于$0 \le n < 2^{32}$，有$0 \le n/(3 \times 2^{33}) < 1/3$，所以根据定理D4，有$q=\lfloor n/3 \rfloor$。

在余数的计算中，如果我们把操作数看成是带符号整数，那么立即乘可能产生溢出，但是，如果把操作数和结果看成是无符号的，那么立即乘不会产生溢出。同样，减也不产生溢出，因为结果在0到2的范围之内，所以余数是正确的。

2. 除以7的无符号除法

对于32位计算机上的除以7的无符号除法，乘数$(2^{32}+3)/7$、$(2^{33}+6)/7$和$(2^{34}+5)/7$全都不合适，因为它们给出过大的误差项。乘数$(2^{35}+3)/7$合适，但是它太大，无法表示成32位字。我们可以用乘以$(2^{35}+3)/7-2^{32}$来代替与这个大数的相乘，再插入一个add指令来修改这个积。代码是:

```
li    M,0x24924925  Magic num, (2**35+3)/7 - 2**32.
mulhu q,M,n         q = floor(M*n/2**32).
add   q,q,n         Can overflow (sets carry).
shrxi q,q,3         Shift right with carry bit.

muli  t,q,7         Compute remainder from
sub   r,n,t         r = n - q*7.
```

这里出现一个问题: add有可能产生溢出。为了允许溢出产生，我们引入一个新指令: 扩展立即右移位（shrxi）指令，这一指令把由add生成的进位和寄存器q的32位量当作一个33位量，并把这个量向右移位，用0填充。在Motorola 68000系列，这一工作可以用2个指令完成: 向右扩展循环移位1个位，再跟一个逻辑右移位3个位（roxr实际上使用X位，但是add指令把X位设置成进位位）。在大多数计算机上，这一工作需要更多指令。例如，在PowerPC上，这需要3个指令: 清除q的最右边的3个位，把进位加到q上，向右循环移位3个位。

在我们实现shrxi指令后，上面的代码计算:

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) / 2^3 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{3n}{7 \times 2^{35}} \right\rfloor$$

对于$0 \leq n < 2^{32}$，有$0 \leq 3n/(7 \times 2^{35}) < 1/7$，所以根据定理D4，有$q = \lfloor n/7 \rfloor$。

Granlund和Montgomery[GM]有一个回避shrxi指令的聪明设计。这一设计所需要的指令数目与上面讨论的shrxi的3指令序列相同，但是，它只使用几乎所有计算机都有的初等指令，而且不会产生溢出。这一设计使用了下面的等式

$$\left\lfloor \frac{q+n}{2^p} \right\rfloor = \left\lfloor \left(\left\lfloor \frac{n-q}{2} \right\rfloor + q \right) / 2^{p-1} \right\rfloor, \quad p \geq 1 \qquad 179$$

把这一等式运用到我们的问题上，其中，$0 \leq M < 2^{32}$，$q = \lfloor Mn/2^{32} \rfloor$，减法不会产生溢出，因为

$$n - q = n - \left\lfloor \frac{Mn}{2^{32}} \right\rfloor = \left\lfloor n - \frac{Mn}{2^{32}} \right\rfloor = \left\lfloor n\left(1 - \frac{M}{2^{32}}\right) \right\rfloor$$

所以显然有$0 \leq n-q < 2^{32}$。同样，加法不会产生溢出，因为

$$\left\lfloor \frac{n-q}{2} \right\rfloor + q = \left\lfloor \frac{n-q}{2} + q \right\rfloor = \left\lfloor \frac{n+q}{2} \right\rfloor$$

而且$0 \leq n$，$q < 2^{32}$。

运用这一思路，我们得到下面除以7的无符号除法的代码：

```
li    M,0x24924925  Magic num, (2**35+3)/7 - 2**32.
mulhu q,M,n         q = floor(M*n/2**32).
sub   t,n,q         t = n - q.
shri  t,t,1         t = (n - q)/2.
add   t,t,q         t = (n - q)/2 + q = (n + q)/2.
shri  q,t,2         q = (n+Mn/2**32)/8 = floor(n/7).

muli  t,q,7         Compute remainder from
sub   r,n,t         r = n - q*7.
```

为了使这一代码正常运行，假设shrxi指令的移位量必须大于0。事实表明，如果$d>1$，乘数$m \geq 2^{32}$（因此需要shrxi指令），那么移位量大于0。

10.9 除数≥1的无符号除法

给定一个字长$W \geq 1$和一个除数d，$1 \leq d < 2^W$，我们希望找到最小整数m和p，使得对任意的$0 \leq n < 2^W$，都有

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \tag{10-22}$$

其中，$0 \leq m < 2^{W+1}$，$p \geq W$。

在无符号的情况下，魔术数M由下式给出：

$$M = \begin{cases} m, & \text{如果 } 0 \leqslant m < 2^W \\ m - 2^W, & \text{如果 } 2^W \leqslant m < 2^{W+1} \end{cases}$$

180

因为等式(10-22)对于$n=d$一定成立，有$\lfloor md/2^p \rfloor = 1$或

$$\frac{md}{2^p} \geqslant 1 \tag{10-23}$$

同带符号的情况一样，设n_c是使得$\mathrm{rem}(n_c, d)=d-1$的n的最大值。它可以由$n_c = \lfloor 2^W/d \rfloor d-1 = 2^W - \mathrm{rem}(2^W, d) - 1$求得。那么有

$$2^W - d \leqslant n_c \leqslant 2^W - 1 \tag{10-24a}$$

和

$$n_c \geqslant d - 1 \tag{10-24b}$$

由这些不等式可得$n_c \geqslant 2^W - 1$。

因为对于$n=n_c$，不等式(10-22)一定成立，所以有

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

或

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}$$

将这个不等式与式(10-23)结合，得到

$$\frac{2^p}{d} \leqslant m < \frac{2^p}{d} \times \frac{n_c + 1}{n_c} \tag{10-25}$$

因为m是满足式(10-25)的最小整数，它是大于或等于$2^p/d$的下一个整数，也就是说，

$$\boxed{m = \frac{2^p + d - 1 - \text{rem}(2^p - 1, d)}{d}} \tag{10-26}$$

将这一不等式与式(10-25)的右边结合并简化，得

$$\boxed{2^p > n_c(d - 1 - \text{rem}(2^p - 1, d))} \tag{10-27}$$

181

1.（无符号）算法

这样，算法就是通过试凑法找满足式(10-27)且大于等于W的最小p。然后，再根据式(10-26)计算m。这个值是满足式(10-22)且$p \geqslant W$的m的最小值。与带符号的情况相同，如果对某个p式(10-27)为真，那么它对所有的更大的p也为真。证明本质上与定理DC1的证明相同，只是在证明中使用了定理D5(b)而不是D5(a)。

2.（无符号）算法可行性证明

我们必须证明式(10-27)总有满足$0 < m < 2^{W+1}$的解。

因为对任何非负整数x，存在大于x而小于或等于$2x+1$的2的幂整数，由式(10-27)，有

$$n_c(d - 1 - \text{rem}(2^p - 1, d)) < 2^p \leqslant 2n_c(d - 1 - \text{rem}(2^p - 1, d)) + 1$$

因为$0 \leqslant \text{rem}(2^{p-1}, d) \leqslant d-1$，有

$$1 \leqslant 2^p \leqslant 2n_c(d - 1) + 1 \tag{10-28}$$

因为n_c、$d < 2^W - 1$，上面表达式变成

$$1 \leqslant 2^p \leqslant 2(2^W - 1)(2^W - 2) + 1$$

或

$$0 \leqslant p \leqslant 2W \tag{10-29}$$

因此，式(10-27)总有解。

如果不强令p等于W，那么由式(10-25)和(10-28)，有

$$\frac{1}{d} \leqslant m < \frac{2n_c(d - 1) + 1}{d} \times \frac{n_c + 1}{n_c}$$

$$1 \leqslant m < \frac{2d - 2 + 1/n_c}{d}(n_c + 1)$$

$$1 \leqslant m < 2(n_c + 1) \leqslant 2^{W+1}$$

如果强令p等于W，那么由式(10-25)，有

$$\frac{2^W}{d} \leqslant m < \frac{2^W}{d} \times \frac{n_c + 1}{n_c}$$

182

因为$1 < d < 2^W - 1$且$n_c \geqslant 2^{W-1}$，有

$$\frac{2^W}{2^W - 1} \leqslant m < \frac{2^W}{1} \times \frac{2^{W-1} + 1}{2^{W-1}}$$

$$2 \leqslant m \leqslant 2^W + 1$$

因此，在任何情况下，m都在上面所描述的“无符号除以7”示例的代码的限制之内。

3.（无符号）积的正确性证明

我们必须证明：如果p和m是由式(10-27)和(10-26)计算而来的，那么它们满足式(10-22)。

容易看出，等式(10-26)和不等式(10-27)蕴含不等式(10-25)。不等式(10-25)与(10-4)几乎相同，而且证明中的余数与$n > 0$的带符号除法的证明中的余数的情况几乎相同。

10.10 并入编译器（无符号）

基于上面证明中所用表达式的直接计算的算法实现起来有一定困难。尽管上面已经证明$p < 2W$，但可能发生$p=2W$的情况（例如，当$W > 4$且$d=2^W-2$时）。当$p=2W$时，很难计算m，因为式(10-26)中的被除数不能用$2W$位的字表示。

然而，通过算法*magic*的“递增除和余数”技术可以实现这一算法。图10-2给出了$W=32$时的算法。这一算法通过回传指示器a来告知是否要生成一个add指令。（在带符号除法的情况下，两数调用通过M和d是否具有相反的符号来判别是否需要生成加指令）。

下面是理解这一算法的一些要点：

- 在几个地方可能产生无符号溢出，它们将被忽略。
- $n_c = 2^W - \mathrm{rem}(2^W, d) - 1 = (2^W - 1) - \mathrm{rem}(2^W - d, d)$。
- 不能用算法*magic*中的方法更新2^p除以n_c的商和余数，因为这里量2*r1可能产生溢出。因此，这一算法需要使用测试“if(r1>=nc-r1),”——尽管“if(2*r1>=nc)”会更自然一些。类似的说明也适用于计算2^p-1除以d的商和余数。
- $0 \leqslant \delta \leqslant d-1$，所以δ可表示成32位无符号整数。

183

- $m = (2^p + d - 1 - \mathrm{rem}(2^p - 1, d))/d = \lfloor (2^p - 1)/d \rfloor + 1 = q_2 + 1$。
- 当魔术数M大于2^W-1时，$M - 2^W$的减法在程序中没有明示；当q2的计算产生溢出时发生这种情况。
- “加”指示器magu.a不能通过M与2^{32}的比较来直接设置，也不能通过q2与$2^{32}-1$的比较来设置，因为会产生溢出。相反，算法在溢出发生之前测试q2。一旦q2达到$2^{32}-1$，则M将大于或等于2^{32}，这时“加”指示器magu.a被设置为1。如果q2小于$2^{32}-1$，那么magu.a将停留在它的初始值0处。
- 不等式(10-27)等价于$2^p/n_c > \delta$。
- 循环测试需要条件“p<64”，因为如果没有这个条件，q1的溢出将会使算法过多地循环，从而给出不正确的结果。

10.11 其他论题（无符号）

定理DC2u 如果不强制p等于W，那么最小乘数m是奇数。

```
struct mu {unsigned M;      // Magic number,
          int a;            // "add" indicator,
          int s;};          // and shift amount.

struct mu magicu(unsigned d) {
                            // Must have 1 <= d <= 2**32-1.
   int p;
   unsigned nc, delta, q1, r1, q2, r2;
   struct mu magu;

   magu.a = 0;              // Initialize "add" indicator.
   nc = -1 - (-d)%d;        // Unsigned arithmetic here.
   p = 31;                  // Init. p.
   q1 = 0x80000000/nc;      // Init. q1 = 2**p/nc.
   r1 = 0x80000000 - q1*nc;// Init. r1 = rem(2**p, nc).
   q2 = 0x7FFFFFFF/d;       // Init. q2 = (2**p - 1)/d.
   r2 = 0x7FFFFFFF - q2*d; // Init. r2 = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (r1 >= nc - r1) {
         q1 = 2*q1 + 1;             // Update q1.
         r1 = 2*r1 - nc;}           // Update r1.
      else {
         q1 = 2*q1;
         r1 = 2*r1;}
      if (r2 + 1 >= d - r2) {
         if (q2 >= 0x7FFFFFFF) magu.a = 1;
         q2 = 2*q2 + 1;             // Update q2.
         r2 = 2*r2 + 1 - d;}        // Update r2.
      else {
         if (q2 >= 0x80000000) magu.a = 1;
         q2 = 2*q2;
         r2 = 2*r2 + 1;}
      delta = d - 1 - r2;
   } while (p < 64 &&
            (q1 < delta || (q1 == delta && r1 == 0)));

   magu.M = q2 + 1;         // Magic number
   magu.s = p - 32;         // and shift amount to return
   return magu;             // (magu.a was set above).
}
```

图10-2 计算无符号除法的魔术数

定理DC3u 如果不强制p等于W，那么对于给定的除数d，存在惟一的对应于最小值p的乘数m。

这些定理的证明与带符号的相应定理的证明非常相似。

1. 具有最优代码的除数（无符号）

对于无符号除法，为了找到具有2指令（li,mulhu）求商最优代码的除数（如果有的话），我们做与带符号除法类似的分析（参见10.7节的“具有最优代码的除数”）。结论是，除d=1的

情况之外，这样的除数是2^W或2^W+1的因子。对于通常的字长，无符号除法很少有具有最优代码的非平凡除数。当$W=16$时，没有这样的除数。当$W=32$时，只有两个这样的除数：641和6 700 417。当$W=64$时，也只有两个这样的除数：274 177和67 280 421 310 721。

值得对$d=2^k$（$k=1,2,...$）特别关注。在这种情况下，算法*magicu*产生$p=W$（强令）和$m=2^{32-k}$。这是m的最小值，但它不是M的最小值。如果进行充分的简化，那么对$p=W+k$会有更好的代码。于是有，$m=2^W$、$M=0$、$a=1$和$s=k$（译者注：原文此处为$s-k$）。生成的代码包含1个乘以0的乘
184 法，并可以简化成1个右移位k个位的指令。作为一种可行的做法，可以认为2的幂的除数是不使用*magicu*的特殊情况。（带符号除法中不会出现这一现象，因为对于带符号除法，m不可能是2的幂。证明：对于$d>0$，不等式(10-4)与(10-3b)结合，可以导出$d-1<2^p/m<d$。因此，$2^p/m$
185 不是整数。对于$d<0$，类似地可以由式(10-16)与(10-15b)结合导出结果）。

对于无符号除法，如果计算机没有shrxi指令，那么$m \geq 2^W$时的代码要比$m<2^W$时的代码糟糕得多。因此，对大乘数出现的频率的了解是非常有益的。对于$W=32$，小于100的整数中有31个“坏”除数：1、7、14、19、21、27、28、31、35、37、38、39、42、45、53、54、55、56、57、62、63、70、73、74、76、78、84、90、91、95和97。

2. 用带符号乘代替无符号乘及用无符号乘代替带符号乘

如果计算机中没有mulhu但是有mulhs（或带符号长乘法），那么8.3节“无符号积高阶位与带符号积高阶位间的转换”所给出的技巧对除以常量的无符号除法仍然有用。

在8.3节，我们给出了从mulhs得到mulhu的7指令序列。然而，因为魔术数M是已知的，所以对于上面的应用它可以简化。因此，编译器可以检测魔术数的最大有效位，并为操作“mulhu q, M, n.”生成如下的代码。这里t表示一个临时寄存器。

```
  M31 = 0          M31 = 1
mulhs q,M,n      mulhs q,M,n
shrsi t,n,31     shrsi t,n,31
and   t,t,M      and   t,t,M
add   q,q,t      add   t,t,n
                 add   q,q,t
```

将和mulhu一同使用的其他指令计算在内，在没有无符号乘法的计算机上，这一操作总共需要6到8个指令来得到除以常量的无符号除法的商。

这一技巧也可颠倒过来，即，使用mulhu来得到mulhs。代码与上面的代码相同，只是mulhs变成mulhu，而且每一行最后的add变成sub。

3. 一个更简单的算法（无符号）

不要求魔术数最小时可以构造一个更简单的算法。可以用下面的表达式代替式(10-27)

$$2^p \geq 2^W(d - 1 - \mathrm{rem}(2^p - 1, d)) \tag{10-30}$$

然后，同前一样，使用式(10-26)计算m。

很显然，这个算法是形式上正确的（也就是说，通过计算得到的m值的确满足式(10-22)），因为这一算法与前面的算法惟一不同的是：对于某些d的值，p的值未必是大整数。可以证明，从式(10-30)和(10-26)计算而得到的m的值小于2^{W+1}。这里省略证明，只给出这一算法（见图
186 10-3）。

```
struct mu {unsigned M;      // Magic number,
          int a;            // "add" indicator,
          int s;};          // and shift amount.

struct mu magicu2(unsigned d) {
                            // Must have 1 <= d <= 2**32-1.
   int p;
   unsigned p32, q, r, delta;
   struct mu magu;
   magu.a = 0;              // Initialize "add" indicator.
   p = 31;                  // Initialize p.
   q = 0x7FFFFFFF/d;        // Initialize q = (2**p - 1)/d.
   r = 0x7FFFFFFF - q*d;    // Init. r = rem(2**p - 1, d).
   do {
      p = p + 1;
      if (p == 32) p32 = 1;      // Set p32 = 2**(p-32).
      else p32 = 2*p32;
      if (r + 1 >= d - r) {
         if (q >= 0x7FFFFFFF) magu.a = 1;
         q = 2*q + 1;            // Update q.
         r = 2*r + 1 - d;        // Update r.
      }
      else {
         if (q >= 0x80000000) magu.a = 1;
         q = 2*q;
         r = 2*r + 1;
      }
      delta = d - 1 - r;
   } while (p < 64 && p32 < delta);
   magu.M = q + 1;          // Magic number and
   magu.s = p - 32;         // shift amount to return
   return magu;             // (magu.a was set above).
}
```

图10-3 求魔术数的简化算法，无符号版本

Alverson[Alv]给出了一个更简单的算法（下一节将讨论这一算法），但是这一算法给出稍大些的m值。算法*magicu2*的特点是，当$d < 2^{W-1}$时，它几乎总是给出最小的m值。对于W=32，*magicu2*不能给出最小乘数的最小的除数是d=102 807，对于这个除数，*magicu*计算m=2 737 896 999，而*magicu2*计算得到的m=5 475 793 997。

对于除数为正的带符号除法也有与*magicu2*类似的算法，但是，这一算法不能对所有除数的带符号除法正确工作。 187

10.12 模除法和地板除法的适用性问题

把除以常量的模除法和地板除法转换成乘法似乎很简单，转换中“如果被除数为负则加1”的步骤可以省去。然而，情况并非如此。上面所给的方法不能以显然的方式运用到模除

法和地板除法。也许在这里有一些工作可做；包括可能要根据被除数的符号对乘数m做一些修改。

10.13 类似的方法

我们可以不通过编写算法*magic*，而是通过给出一个列出一些小除数的魔术数和移位量的表，来计算除法。对于一个除数d，如果它等于列在表中的某个小除数乘以2的某个幂，那么可以对它做如下处理：

1) 计数d的后缀0数目，并用k表示之。

2) 以$d/2^k$(右移位k个位)为查表的参数。

3) 使用在表中找到的魔术数。

4) 使用在表中找到的移位量加k。

因此，如果表格中包含除数3、5、25等等，那么可以处理除数6、10、100等等。

这一过程通常给出最小的魔术数，但并非总给出最小魔术数。对于$W=32$，不能给出最小魔术数的最小正除数是$d=334\ 972$，对于这一除数，$m=3\ 361\ 176\ 179$及$s=18$。然而，对于$d=334\ 972$的最小魔术数是$m=840\ 294\ 045$且$s=16$。同样，这一过程也不能给出$d=-6$时的最小魔术数。在上面的两种情况中，输出代码的质量会受到影响。

据作者所知，Alverson[Alv]首先论述了上面的方法对所有除数都能完全精确地完成工作。使用我们的记法，他的除以d的无符号除法的方法是，设置移位量$p=W+\lceil \log_2 d \rceil$，乘数$m=\lceil 2^p/d \rceil$，然后再通过$n \div d = \lfloor mn/2^p \rfloor$（也就是乘和右移位）来完成除法运算。他证明了，乘数m小于2^{W+1}，以及这一方法对于所有可以用W位表示的n都能得到正确无误的商。

Alverson的方法比我们的方法要简单，在他的方法中不需要使用试凑法来决定p，因此这一方法更适合于构建硬件，这正是他最初的兴趣。然而，他的乘数m总是大于或等于2^W，因此，作为软件应用，它总是给出上述的“除以7”的例子所示的代码（也就是说，总有add和shrxi，或相应的4个指令）。因为大多数的小除数都可以用小于2^W的乘数来处理，寻找这些乘数似乎很值得。

对于带符号除法，Alverson提议寻找$|d|$对应的乘数和一个字长为$W-1$的字m（那么有$2^{W-1} < m < 2^W$），用m乘以被除数，如果操作数符号相反，那么取结果的负。（当被除数是2^{W-1}时，乘数也必须给出正确的结果，即，最大负数的绝对值。）乘数$m > 2^W$时，这一建议给出的代码似乎要比这里给出的代码更好。将这一建议运用于除以7的带符号除法，给出如下的代码，在此，我们使用了关系$-x=\bar{x}+1$来回避分支：

188

```
abs    an,n
li     M,0x92492493  Magic number, (2**34+5)/7.
mulhu  q,M,an        q = floor(M*an/2**32).
shri   q,q,2
shrsi  t,n,31        These three instructions
xor    q,q,t         negate q if n is
sub    q,q,t         negative.
```

这一代码没有我们给出的除以7的带符号除法的代码好（6个指令对7个指令），但是在有abs和mulhu而没有mulhs的计算机上，这一代码会有用。

10.14 魔术数示例

表10-1 W=32时的若干魔术数

d	带符号		无符号		
	M（十六进制）	s	M（十六进制）	a	s
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000001	k-1	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

189

表10-2 W=64时的若干魔术数

d	带符号		无符号		
	M（十六进制）	s	M（十六进制）	a	s
-5	99999999 99999999	1			
-3	55555555 55555555	1			
-2^k	7FFFFFFF FFFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000000 00000001	k-1	2^{64-k}	0	0
3	55555555 55555556	0	AAAAAAAA AAAAAAAB	0	1
5	66666666 66666667	1	CCCCCCCC CCCCCCCD	0	2
6	2AAAAAAA AAAAAAAB	0	AAAAAAAA AAAAAAAB	0	2
7	49249249 24924925	1	24924924 92492493	1	3
9	1C71C71C 71C71C72	0	E38E38E3 8E38E38F	0	3
10	66666666 66666667	2	CCCCCCCC CCCCCCCD	0	3
11	2E8BA2E8 BA2E8BA3	1	2E8BA2E8 BA2E8BA3	0	1
12	2AAAAAAA AAAAAAAB	1	AAAAAAAA AAAAAAAB	0	3
25	A3D70A3D 70A3D70B	4	47AE147A E147AE15	1	5
125	20C49BA5 E353F7CF	4	0624DD2F 1A9FBE77	1	7
625	346DC5D6 3886594B	7	346DC5D6 3886594B	0	7

10.15 除以常数的精确除法

“精确除法”的意思就是，已经通过某种手段预先知道这一除法的余数是0的除法。尽管这种情况不普遍，但是这种情况的确可能发生，例如，在C语言中，当我们做两个指针的减法时就会发生这种情况。在C语言中，当p和q是指针时，仅当p和q指向同一数组的对象时，$p-q$的结果才有定义且是可移植的[H&S, 7.6.2节]。如果数组中的元素大小是s，那么差$p-q$的目标代码计算$(p-q)/s$。

本节的内容是受[GM，第9节]的启发而得的。

我们要给出的方法同时适用于带符号和无符号精确除法，它基于下面的定理。

定理M1 如果a和m是互素的整数，那么存在一个整数$\bar{a}$，使得$1 \leqslant \bar{a} < m$，且

$$a\bar{a} \equiv 1 \pmod{m}$$

因此，$\bar{a}$是a模m的乘法逆元素。有几种证明这个定理的方法；[MZM，52]给出三个证明。

190 下面的证明只需要对同余有基本的了解。

证明 我们将证明比定理更一般一些的结论。如果a和m是互素（因此是非零的），那么，当x取遍模m的值时，ax模m取遍所有m个不同的值。例如，如果$a=3$，$m=8$，那么当x取遍0到7的值时，ax =0、3、6、9、12、15、18、21，或模8后值减小到ax=0、3、6、1、4、7、2、5。注意，所有从0到7的值在最后的序列中都出现。

为了给出这一论断的一般证明，我们假设它不是真的。那么就存在不同的整数，它们乘以a后映射到相同的值；也就是说，存在x和y，且$x \not\equiv y \pmod{m}$，使得

$$ax \equiv ay \pmod{m}$$

但是，这样就存在一个整数k，使得

$$ax - ay = km \quad 或$$

$$a(x-y) = km$$

因为a和m没有公因子，$x-y$一定是m的倍数，也就是说，

$$x \equiv y \pmod{m}$$

这与假设矛盾。

现在，当x取遍所有0到$m-1$的m个值时，因为ax模m取所有不同的值，所以存在x，$ax \equiv 1 \pmod{m}$。

由证明可知，存在（模m）惟一的x，使得$ax \equiv 1 \pmod{m}$，也就是说，不计附加的m的倍数，乘法逆元素是惟一的。这也表明，对于任意的整数b，(模m)存在惟一整数x，使得$ax \equiv b \pmod{m}$。

例如，考虑m=16的情况。那么$\bar{3}$=11，因为$3 \times 11 = 33 \equiv 1 \pmod{16}$。我们也可以取$\bar{3}=-5$，因为$3 \times (-5) = -15 \equiv 1 \pmod{16}$。类似地，$-\bar{3}$=5，因为$(-3) \times 5 = -15 \equiv 1 \pmod{16}$。

这些观察非常重要，因为它们表明这一原理可以运用于带符号与无符号数。如果我们正工作在一台4位计算机的无符号整数环境下，那么我们取$\bar{3}$=11。而在带符号的情况下，我们取$\bar{3}=-5$。但是，11和−5有相同的2的补码表示（因为它们的差是16），所以同一个计算机字的内容可以在两个环境下充当乘法逆元素。

这一定理可以直接应用于W位计算机上的除以奇整数d的（带符号和无符号）除法的问题。因为任意奇整数与2^W互素，这一定理说，如果d是奇数，那么存在整数$\bar{d}$（在范围0到2^W-1或-2^{W-1}到$2^{W-1}-1$内是惟一的），使得

$$d\bar{d} \equiv 1 \pmod{2^W} \tag{191}$$

因此，如果整数n是d的倍数，则有

$$\frac{n}{d} \equiv \frac{n}{d}(d\bar{d}) \equiv n\bar{d} \pmod{2^W}$$

换句话说，n/d的计算可以由n乘以$\bar{d}$然后保留积的最右边的W位得到。

如果除数d是偶数，设$d=d_0 \cdot 2^k$，这里d_0是奇数且$k \geqslant 1$。那么，简单地把n右移位k个位（移出0位），再乘以$\bar{d}_0$（也可以在乘运算之后做右移位）。

下面是n除以7的代码，这里的n是7的倍数。无论所考虑的是带符号还是无符号除法，这一代码都给出正确结果。

```
li      M,0xB6DB6DB7   Mult. inverse, (5*2**32 + 1)/7.
mul     q,M,n          q = n/7.
```

1. 利用欧几里德算法计算乘法逆元素

如何求乘法逆元素呢？标准方法是通过“扩展的欧几里德算法”。下面的讨论主要是针对我们的问题，有兴趣的读者可以查看[NZM，13]和[Knu2，4.5.2节]，那里可以看到更全面的讨论。

给定一个奇除数d，我们通过解下式来求x

$$dx \equiv 1 \pmod{m}$$

其中，在我们的应用中，$m=2^W$、W是计算机的字长。只要求满足下式的整数x和y（正整数、负整数或0）即可。

$$dx + my = 1$$

为了实现这一任务，我们首先将d变成正的，也就是在d上加上一个足够大的m的倍数。（d和$d+km$有相同的乘法逆元素。）其次，写出下面的等式（其中，d、$m>0$）：

$$d(-1) + m(1) = m - d \quad \text{(i)}$$
$$d(1) + m(0) = d \quad \text{(ii)}$$

如果$d=1$，因为等式(ii)表明$x=1$，我们就完成任务了。否则，计算

$$q = \left\lfloor \frac{m-d}{d} \right\rfloor \tag{192}$$

第三步、把等式(ii)的两边乘以q，然后再用等式（i）减去这一新的等式，得到

$$d(-1-q) + m(1) = m - d - qd = \text{rem}(m-d, d)$$

这一等式成立，因为我们只是用一个常量乘以一个等式，再从另一个等式中减去这一等式。如果$\text{rem}(m-d,d)=1$，我们就完成任务了；最后的等式就是解，而且$x=-1-q$。

对最后两个等式重复这一过程，得到第四个等式，这样继续下去，直到这一等式的右边等于1为止。经模m简化后的d的乘数就是所求的d的乘法逆元素。

顺便提一下，如果$m-d<d$，那么第一个商是0，因此第三行将与第一行相同，所以第二个商是非零的。另外，大多数教科书都以下式作为测试的第一行：

$$d(0) + m(1) = m$$

但在我们的应用中，$m=2^W$在计算机中是不能表示的。

这一过程可以用一个例子来很好地说明。设$m=256$和$d=7$，那么计算过程如下。为了得到第三行，注意$q=\lfloor 249/7 \rfloor = 35$。

```
 7(-1) + 256( 1) = 249
 7( 1) + 256( 0) = 7
7(-36) + 256( 1) = 4
7( 37) + 256(-1) = 3
7(-73) + 256( 2) = 1
```

因此，7的乘法逆元素模256是−73、或用0到255范围内的数表示为183。验证：$7 \times 183 = 1\,281 \equiv 1 \pmod{256}$。

从第三行开始，右边这一列的整数都是对应于把它上面的数当作除数时的余数（d是被除数），所以，这些余数形成一个严格递减的非负整数序列。因此，这个序列一定结束于0（就如上面的例子再进行一步那样）。另外，0之前的数一定是1。理由如下：假设序列在0前的值是b，且$b \neq 1$；那么，b前面的整数一定是b的倍数，我们设这个倍数是$k_1 b$，因为下一个余数是0。$k_1 b$前边的数一定是$k_1 k_2 b + b$、因为下一个余数是b。持续这一序列，每个数都一定是b的倍数，包含前两个数（在上面的例子中就是249和7）。但是，这是不可能的，因为前两个数是$m-d$和d、它们是互素的。

这构成上面的过程一定终止而且右边一栏的最后值是1的一个非形式化证明。因此可以通过它来找d的乘法逆元素。 193

为了在计算机上实现这一过程，首先注意，如果$d<0$，我们应该把2^W加到d上。但是，由于2的补码算术，在这里没有必要做任何事情，只要简单地解释d为一个无符号数即可，不用考虑对它的应用是如何解释的。

q的计算必须使用无符号除法。

注意、可以用模m来进行余数序列的计算，因为这不会改变上述等式右侧的余数序列的值（这些值都在0到$m-1$之间）。这非常重要，因为这允许计算使用“单精度”，运用计算机的模2^W无符号算术来实现这一计算。

这个表中的大多数量都不必表示出来。256的倍数一栏不必表示，因为在解$dx+my=1$中，我们不需要y的值。没有必要表示第一栏中的d。于是，经过精简，上面例子的计算可以表示如下：

```
255   249
  1     7
220     4
 37     3
183     1
```

实现这一计算的C语言算法如图10-4所示。

循环继续的条件是(v2>1)而不是更自然的(v2!=1)，其原因是如果使用后者，那么当这一算法的参数是偶数的话，循环就不会停止。即使参数被误用，程序最好也不要进行无穷

循环。（如果参数d是偶数，那么v2不会取到值1，但它终究会变成0。）

```
unsigned mulinv(unsigned d) {              // d must be odd.
   unsigned x1, v1, x2, v2, x3, v3, q;

   x1 = 0xFFFFFFFF;     v1 = -d;
   x2 = 1;              v2 = d;
   while (v2 > 1) {
      q = v1/v2;
      x3 = x1 - q*x2;   v3 = v1 - q*v2;
      x1 = x2;          v1 = v2;
      x2 = x3;          v2 = v3;
   }
   return(x2);
}
```

图10-4 通过欧几里德算法求模2^{32}的乘法逆元素

194

当给定一个偶数参数，这个算法计算的是什么呢？它求的是满足$dx \equiv 0 \pmod{2^{32}}$的$x$，这也许没有用处。然而，稍加修改，使循环的条件为(v2!=0)，返回x1而不是返回x2，那么它就计算满足$dx \equiv g \pmod{2^{32}}$的$x$，这里$g$是$d$和$2^{32}$的最大公约数，也就是整除$d$的最大的2的幂。修改后的算法仍能求得奇数$d$的乘法逆元素，但是，这时的算法比修改前的算法多需要一次循环。

对于上面算法所需要的循环次数，当d是奇数而且小于20时，它需要的最大循环次数是3，平均是1.7。当d在1000左右时，它需要的循环的最大次数是11，平均数大约是6。

2. 用牛顿方法计算乘法逆元素

众所周知，对于任意的非零实数$d \neq 0$，只要给出充分接近$1/d$的初始值x_0，就可以通过迭代计算下面的式子给出$1/d$的任意精确的值：

$$x_{n+1} = x_n(2 - dx_n) \tag{10-31}$$

每次迭代都使精确的数字的数目提高大约一倍。

鲜为人知的是，这一公式可以用来在整数的模算术内寻找乘法逆元素！例如，为了寻找模256下的3的乘法逆元素，以x_0=1（任意奇数都可以）为初始值。然后，有

$$x_1 = 1(2 - 3\times 1) = -1$$
$$x_2 = -1(2 - 3(-1)) = -5$$
$$x_3 = -5(2 - 3(-5)) = -85$$
$$x_4 = -85(2 - 3(-85)) = -21845 \equiv -85 \pmod{256}$$

迭代达到了一个模256的不动点，所以，-85或171是3（模256）的乘法逆元素。所有计算都是模256的。

为什么这一方法是可行的？因为，如果x_n满足

$$dx_n \equiv 1 \pmod{m}$$

而且x_{n+1}由等式(10-31)定义，那么有

$$dx_{n+1} \equiv 1 \pmod{m^2}$$

195

为了证明这一事实，设$dx_n=1+km$，那么，

$$\begin{aligned} dx_{n+1} &= dx_n(2-dx_n) \\ &= (1+km)(2-(1+km)) \\ &= (1+km)(1-km) \\ &= 1-k^2m^2 \\ &\equiv 1 \pmod{m^2} \end{aligned}$$

在我们的应用中，m是2的幂，即2^N。在这种情况下，如果

$$dx_n \equiv 1 \pmod{2^N}, \text{ 那么}$$

$$dx_{n+1} \equiv 1 \pmod{2^{2N}}$$

在某种意义上说，如果x_n是$\bar{d}$的近似值，那么，等式(10-31)的每次迭代就把近似值的“精确度”的位数提高一倍。

非常巧合地，对于模8，每个（奇数）d的乘法逆元素都是它本身。因此，$x_0=d$是$\bar{d}$的合理、简单的初始猜测值。这时，(10-31)将给出$x_1, x_2, \ldots$，使得

$$dx_1 \equiv 1 \pmod{2^6}$$

$$dx_2 \equiv 1 \pmod{2^{12}}$$

$$dx_3 \equiv 1 \pmod{2^{24}}$$

$$dx_4 \equiv 1 \pmod{2^{48}}, \text{ 等等}$$

因此，四次迭代就足以找到模2^{32}的乘法逆元素（如果$x \equiv 1 \pmod{2^{48}}$，那么对所有的$n<48$，$x \equiv 1 \pmod{2^n}$）。这就导出图10-5的C程序，其中，所有计算都是模2^{32}的。

对于大约一半的d的值，这一程序需要4.5次迭代，或9个乘法。对于另一半的值（它们的x_n的初始值“精确到4个位”，也就是说，$d^2 \equiv 1 \pmod{16}$），算法需要7个或更少的乘法，通常是7个乘法。因此，平均需要8个乘法。

```
unsigned mulinv(unsigned d) {        // d must be odd.
   unsigned xn, t;

   xn = d;
loop: t = d*xn;
      if (t == 1) return xn;
      xn = xn*(2 - t);
      goto loop;
}
```

196

图10-5 使用牛顿方法的模2^{32}的乘法逆元素

有一个变形，它不管d是什么，只简单地进行四次循环迭代，也可以展开迭代而取消循环控制（8个乘法）。另外一个变形是设法使初始值x_0“精确到4个位”（也就是找满足$dx_0 \equiv 1 \pmod{16}$的x_0）。那么，它只需要3次循环迭代。下面是设置初始估计值的几个方法：

$$x_0 \leftarrow d + 2((d+1) \,\&\, 4), \text{ 及}$$

$$x_0 \leftarrow d^2 + d - 1$$

这里，乘以2的乘法是1个左移位，计算是模2^{32}的（忽略溢出）。因为第二个公式使用了一个乘法，因此只节省1个乘法。

对执行时间的担心当然对于编译器应用是毫无意义的。对于这样的应用，很少使用循环，除非为了节省空间。但是，也可能有的应用希望快速计算乘法逆元素。

3. 乘法逆元素示例

我们以表10-3所示的乘法逆元素示例结束本节。

表10-3 乘法逆元素示例

d	$\bar{d}$		
（十进制）	模 16 （十进制）	模 2^{32} （十六进制）	模 2^{64} （十六进制）
-7	-7	49249249	9249249249249249
-5	3	33333333	3333333333333333
-3	5	55555555	5555555555555555
-1	-1	FFFFFFFF	FFFFFFFFFFFFFFFF
1	1	1	1
3	11	AAAAAAAB	AAAAAAAAAAAAAAAB
5	13	CCCCCCCD	CCCCCCCCCCCCCCCD
7	7	B6DB6DB7	6DB6DB6DB6DB6DB7
9	9	38E38E39	8E38E38E38E38E39
11	3	BA2E8BA3	2E8BA2E8BA2E8BA3
13	5	C4EC4EC5	4EC4EC4EC4EC4EC5
15	15	EEEEEEEF	EEEEEEEEEEEEEEEF
25		C28F5C29	8F5C28F5C28F5C29
125		26E978D5	1CAC083126E978D5
625		3AFB7E91	D288CE703AFB7E91

197

读者可能注意到，在几种情况（d=3、5、9、11）下，d的乘法逆元素和除以d的无符号除法的魔术数相同（参见10.14节）。这多少是一种巧合。对于这些数来说，魔术数M等于乘数m，而且它们具有$(2^p+1)/d$的形式，$p \geq 32$。在这种情况下，有

$$Md = \left(\frac{2^p + 1}{d}\right)d \equiv 1 \pmod{2^{32}}$$

所以有$M \equiv \bar{d} \pmod{2^{32}}$。

10.16 除以常数的除法的零余数检测

除数d的乘法逆元素可以用于除以d的除法的零余数检测 [GM] 。

1. 无符号

首先，考虑除以奇数d的无符号除法。$\bar{d}$表示d的乘法逆元素。那么，因为$d\bar{d} \equiv 1 \pmod{2^W}$，这里$W$是计算机的以位计算的字长，$\bar{d}$也是奇数。因此，$\bar{d}$与$2^W$互素，而且如上节定理MI的证明所示，当$n$取遍所有模$2^W$的$2^W$个不同值时，$n\bar{d}$取遍所有模$2^W$的$2^W$个值。

如上节所示，如果n是d的倍数，则有

$$\frac{n}{d} \equiv n\bar{d} \pmod{2^W}$$

也就是说，对$n=0, d, 2d, \ldots, \lfloor (2^W-1)/d \rfloor$ d有$n\bar{d} \equiv 0, 1, 2, \ldots, \lfloor (2^W-1)/d \rfloor \pmod{2^W}$。因此，对于一个不是$d$的倍数的$n$，$n\bar{d}$的值模$2^W$缩小到0到$2^W-1$的范围之内后一定大于$\lfloor (2^W-1)/d \rfloor$。

这一结果可以用来做零余数检测。例如，如果要检测整数n是否是25的倍数，让n乘以$\bar{25}$，再把它最右边的W位与$\lfloor (2^W-1)/25 \rfloor$相比较。在基本的RISC上，代码为：

```
li      M,0xC28F5C29   Load mult. inverse of 25.
mul     q,M,n          q = right half of M*n.
li      c,0x0A3D70A3   c = floor((2**32-1)/25).
cmpleu  t,q,c          Compare q and c, and branch
bt      t,is_mult      if n is a multiple of 25.
```

198

为了把这一检测扩展到除数为偶数的情况，设$d=d_0 \cdot 2^k$，其中d_0是奇数，且$k>1$。那么，因为一个整数能被d整除当且仅当它既能被d_0整除又能被2^k整除，而且因为n和$n\bar{d}_0$有相同的后缀0数目（$\bar{d}_0$是奇数），n是d的倍数的检测是：

设置$q=\mathrm{mod}(n\bar{d}_0, 2^W)$;

$q < \lfloor (2^W-1)/d_0 \rfloor$且$q$的后缀0的数目至少为$k$。

其中，“mod”函数是为了把$n\bar{d}_0$缩小到区间$[0, 2^W-1]$。

直接实现上面的检测需要两次测试和条件分支，但是，如果计算机有循环移位指令，那么检测的实现可以减少到只需1个比较分支指令。这由下面的定理得出，其中$a \overset{rot}{>>} k$表示把计算机字a向右旋转k个位置（$0<k<32$）。

定理ZRU 当且仅当$x \overset{rot}{>>} k \overset{u}{<} \lfloor a/2^k \rfloor$时，$x \overset{u}{<} a$且$x$的尾部有$k$个0位。

证明 （假设是32位计算机）假设$x \overset{u}{<} a$且x的尾部有k个0位。那么，因为有$x \overset{u}{<} a$，所以有$\lfloor x/2^k \rfloor \overset{u}{<} \lfloor a/2^k \rfloor$。但是，$\lfloor x/2^k \rfloor = x \overset{rot}{>>} k$。因此有$x \overset{rot}{>>} k \overset{u}{<} \lfloor a/2^k \rfloor$。如果$x$的尾部没有$k$个0位，那么$x \overset{rot}{>>} k$就不是从$k$个0位开始的，而$\lfloor a/2^k \rfloor$是从$k$个0位开始的，所以$x \overset{rot}{>>} k \overset{u}{>} \lfloor a/2^k \rfloor$。最后，如果$x \overset{u}{>} a$且$x$的尾部有$k$个0位，那么，由$x$的开始$32-k$个位组成的整数一定大于由$a$的开始$32-k$个位组成的整数，从而满足$\lfloor x/2^k \rfloor \overset{u}{>} \lfloor a/2^k \rfloor$。

运用这个定理，n是d的倍数的检测如下，其中n和d是大于等于1的无符号整数且$d=d_0 \cdot 2^k$，d_0是奇数：

$$q \leftarrow \mathrm{mod}(n\bar{d}_0, 2^W)$$

$$q \overset{rot}{>>} k \overset{u}{\leqslant} \lfloor (2^W-1)/d \rfloor$$

这里，我们使用了$\lfloor \lfloor (2^W-1)/d_0 \rfloor /2^k \rfloor = \lfloor (2^W-1)/(d_0 \cdot 2^k) \rfloor = \lfloor (2^W-1)/d \rfloor$。

例如，下面的代码检测一个无符号整数n，看它是否是100的倍数：

```
li      M,0xC28F5C29   Load mult. inverse of 25.
mul     q,M,n          q = right half of M*n.
shrri   q,q,2          Rotate right two positions.
li      c,x'028F5C28'  c = floor((2**32-1)/100).
```

```
cmpleu t,q,c          Compare q and c, and branch
bt     t,is_mult      if n is a multiple of 100.
```

199

2. 带符号，除数 > 2

对于带符号除法，正如上节所述，如果n是d的倍数，且d是奇数，那么有：

$$\frac{n}{d} \equiv n\bar{d} \pmod{2^W}$$

因此，对于$n = \lceil -2^{W-1}/d \rceil \cdot d, \ldots, -d, 0, d, \ldots, \lfloor (2^{W-1}-1)/d \rfloor \cdot d$，我们有$n\bar{d} \equiv \lceil -2^{W-1}/d \rceil, \ldots, -1, 0, 1, \ldots, \lfloor (2^{W-1}-1)/d \rfloor \pmod{2^W}$。另外，因为$\bar{d}$与$2^W$互素，当$n$取遍模$2^W$的所有$2^W$个不同值时，$n\bar{d}$取遍模$2^W$的所有$2^W$个不同值。因此，$n$是$d$的倍数当且仅当：

$$\lceil -2^{W-1}/d \rceil \leqslant \mathrm{mod}(n\bar{d}, 2^W) \leqslant \lfloor (2^{W-1}-1)/d \rfloor$$

其中，“mod”函数是为了把$n\bar{d}$缩小到区间$[-2^{W-1}, 2^{W-1}-1]$。

因为d是奇数，而且如我们假设的那样，它是不等于1的正数，它不能整除2^{W-1}，所以上面的表达式可以稍加简化为：

$$\lceil -2^{W-1}/d \rceil = \lceil (-2^{W-1}+1)/d \rceil = -\lfloor (2^{W-1}-1)/d \rfloor$$

因此，对于带符号数，n是d的倍数的检测如下，其中，$d = d_0 \cdot 2^k$，且d_0是奇数：

设置q=mod$(n\bar{d}_0, 2^W)$;

$-\lfloor (2^{W-1}-1)/d_0 \rfloor < q < \lfloor (2^{W-1}-1)/d_0 \rfloor$且$q$的尾部至少有$k$个0位。

表面上，这似乎需要3个测试和分支。然而，与无符号的情况相同，通过使用下面的定理，这一检测可以减少到只需要1个比较分支指令。

定理ZRS 如果$a > 0$，则下面的断言是等价的：

(1) $-a < x < a$且x的尾部至少有k个0位，

(2) $\mathrm{abs}(x) \overset{rot}{>>} k \overset{u}{<} \lfloor a/2^k \rfloor$，

(3) $x + a' \overset{rot}{>>} k \overset{u}{<} \lfloor 2a'/2^k \rfloor$。

其中，a'是将a最右边k位设置为0而得到的值（即，$a' = a \& -2^k$）。 200

证明（假设是32位计算机） 为了证明(1)等价于(2)，显然，断言$-a < x < a$等价于$\mathrm{abs}(x) < a$。于是，因为这一不等式两边都是非负的，等价性由定理ZRU可得。

为了证明(1)与(3)等价，注意，断言(1)等价于用a'代替a后的(1)。于是，根据4.1节介绍的边界检测定理，(1)与下式等价：

$$x + a' \overset{u}{\leqslant} 2a'$$

因为当且仅当x的尾部有k个0位时，$x + a'$的尾部才有k个0位，运用定理ZRU可得结果。

使用这一定理的(3)，n是d的倍数的检测如下，其中n和d是大于等于2的带符号整数，且$d = d_0 \cdot 2^k$，d_0是奇数：

$$q \leftarrow \mathrm{mod}(n\bar{d}_0, 2^W)$$
$$a' \leftarrow \lfloor (2^{W-1}-1)/d_0 \rfloor \,\&\, -2^k$$
$$q + a' \overset{rot}{>>} k \overset{u}{\leqslant} \lfloor (2a')/2^k \rfloor$$

(因为d是常量，所以a'可以在编译时计算。)

例如，下面的代码检测一个带符号整数n，看它是否是100的倍数。注意，常量$\lfloor 2a'/2^k \rfloor$可以通过把常量a'右移位$k-1$个位得到，所以可以节省一个指令或在处理比较操作数时节省一个从内存的装入。

```
li      M,0xC28F5C29  Load mult. inverse of 25.
mul     q,M,n         q = right half of M*n.
li      c,0x051EB850  c = floor((2**31 - 1)/25) & -4.
add     q,q,c         Add c.
shrri   q,q,2         Rotate right two positions.
shri    c,c,1         Compute const. for comparison.
cmpleu t,q,c          Compare q and c, and
bt      t,is_mult     branch if n is a mult. of 100.
```

201

我认为无法想象有像除法那么可恶的操作。

答案需要猜测，然后通过乘法来评估。

让我们付出昂贵代价，每天消耗在循环中。

除法的代码常常令人恐惧；长除代码更令人恐怖。

证明让人们的大脑超载，地板和天花板使人们发疯。

202 编写除法的好代码召唤着英雄，但是即使上帝也不能除以零！

第11章　初等函数

11.1　整数平方根

“整数平方根”函数是指函数$\lfloor \sqrt{x} \rfloor$。为了扩展它的应用范围，并且回避对负参数的处理，我们假设x是无符号的。因此有$0 \leqslant x < 2^{32}-1$。

1. 牛顿方法

对于浮点数，它们的平方根几乎都是用牛顿方法来计算的。这一方法首先设法得到$\sqrt{a}$的初始评估值g_0。然后，通过下式得到一系列更精确的评估值：

$$g_{n+1} = \left(g_n + \frac{a}{g_n}\right)/2$$

迭代二次收敛，也就是说，如果某个点g_n精确到n位，那么g_{n+1}精确到$2n$位。程序必须知道什么时候迭代已经充分，从而可以停止迭代。

令人惊喜的是，牛顿方法在整数范围内很有用。为了了解这一点，我们需要下面的定理：

定理　设$g_{n+1}=\lfloor (g_n+\lfloor a/g_n \rfloor)/2 \rfloor$，$g_n$和$a$是大于0的整数，则

(a) 如果$g_n > \lfloor \sqrt{a} \rfloor$，那么$\lfloor \sqrt{a} \rfloor \leqslant g_{n+1} < g_n$，而且

(b) 如果$g_n = \lfloor \sqrt{a} \rfloor$，那么$\lfloor \sqrt{a} \rfloor \leqslant g_{n+1} \leqslant \lfloor \sqrt{a} \rfloor + 1$。

也就是说，如果g_n是$\lfloor \sqrt{a} \rfloor$的过大的估测，那么下一个评估g_{n+1}一定会严格小于前一个估测，而且不小于$\lfloor \sqrt{a} \rfloor$。因此，如果我们以过大的估测开始，那么序列单调收敛。如果估测$g_n = \lfloor \sqrt{a} \rfloor$，那么下一个估测或者等于$g_n$，或者比它大1。这就提供了决定序列什么时候收敛的简单方法：如果我们开始于$g_0 > \lfloor \sqrt{a} \rfloor$，那么当$g_{n+1} \geqslant g_n$时发生收敛，且所求结果就是$g_n$。

然而，对于a=0的情况，必须特殊处理，因为这将导致0除以0。

证明　(a)因为g_n是一个整数，所以有

$$g_{n+1} = \left\lfloor \left(g_n + \left\lfloor \frac{a}{g_n} \right\rfloor\right)/2 \right\rfloor = \left\lfloor \left\lfloor g_n + \frac{a}{g_n} \right\rfloor /2 \right\rfloor = \left\lfloor \left(g_n + \frac{a}{g_n}\right)/2 \right\rfloor = \left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor$$

因为$g_n > \lfloor \sqrt{a} \rfloor$，而且$g_n$是整数，所以$g_n > \sqrt{a}$。由公式$g_n = (1+\varepsilon)\sqrt{a}$定义$\varepsilon$。那么$\varepsilon>0$且有：

$$\left\lfloor \frac{g_n^2 + a}{2g_n} \right\rfloor = g_{n+1} \leqslant \frac{g_n^2 + a}{2g_n}$$

$$\left\lfloor \frac{(1+\varepsilon)^2 a + a}{2(1+\varepsilon)\sqrt{a}} \right\rfloor = g_{n+1} < \frac{g_n^2 + g_n^2}{2g_n}$$

$$\left\lfloor \frac{2 + 2\varepsilon + \varepsilon^2}{2(1+\varepsilon)} \sqrt{a} \right\rfloor = g_{n+1} < g_n$$

$$\left\lfloor \frac{2 + 2\varepsilon}{2(1+\varepsilon)} \sqrt{a} \right\rfloor \leqslant g_{n+1} < g_n$$

$$\lfloor\sqrt{a}\rfloor \leqslant g_{n+1} < g_n$$

所以定理的(a)成立。

(b) 因为$g_n = \lfloor\sqrt{a}\rfloor$，有 $\sqrt{a}-1 < g_n \leqslant \sqrt{a}$，所以$g_n^2 \leqslant a < (g_n+1)^2$。因此有:

$$\left\lfloor \frac{g_n^2 + g_n^2}{2g_n} \right\rfloor \leqslant g_{n+1} \leqslant \left\lfloor \frac{g_n^2 + (g_n+1)^2}{2g_n} \right\rfloor$$

$$\lfloor g_n \rfloor \leqslant g_{n+1} \leqslant \left\lfloor g_n + 1 + \frac{1}{2g_n} \right\rfloor$$

$$\lfloor\sqrt{a}\rfloor \leqslant g_{n+1} \leqslant \lfloor g_n + 1 \rfloor \quad (\text{因为 } g_n \text{ 是整数且 } \frac{1}{2g_n} < 1)$$

$$\lfloor\sqrt{a}\rfloor \leqslant g_{n+1} \leqslant \lfloor g_n \rfloor + 1 = \lfloor\sqrt{a}\rfloor + 1$$

所以定理的(b)成立。

运用牛顿方法计算 $\lfloor\sqrt{x}\rfloor$ 的难点是获得初始估测值。图11-1的算法把初始估测值g_0设置为
204 大于或等于 $\sqrt{x}$ 的最小的2的幂。例如，对于x=4，g_0=2；对于x=5，g_0=4。

```
int isqrt(unsigned x) {
   unsigned x1;
   int s, g0, g1;

   if (x <= 1) return x;
   s = 1;
   x1 = x - 1;
   if (x1 > 65535) {s = s + 8; x1 = x1 >> 16;}
   if (x1 > 255)   {s = s + 4; x1 = x1 >> 8;}
   if (x1 > 15)    {s = s + 2; x1 = x1 >> 4;}
   if (x1 > 3)     {s = s + 1;}

   g0 = 1 << s;                    // g0 = 2**s.
   g1 = (g0 + (x >> s)) >> 1;      // g1 = (g0 + x/g0)/2.

   while (g1 < g0) {               // Do while approximations
      g0 = g1;                     // strictly decrease.
      g1 = (g0 + (x/g0)) >> 1;
   }
   return g0;
}
```

图11-1 整数平法根，牛顿方法

因为初始估测值g_0是2的幂，因此没有必要做除法来得到g_1；做右移位就足够了。

因为初始估测值精确到大约1个位，而且牛顿方法是二次收敛的（每一次迭代精确的位数加倍），人们可能期望算法在大约5次迭代内收敛（在一台32位计算机上），这需要4个除法（因为第一次迭代被右移位指令所替代）。彻底的实验表明，所需除法的最大数目为5，而如果参数的最大值是16 785 407，除法的最大数目为4。

如果前导0数目指令可用的话，那么很容易得到第一个估测值：用下面的代码替换上面过

程的前七行可执行代码。

```
if (x <= 1) return x;
s = 16 - nlz(x - 1)/2;
```

如果前导0数目指令不可用，另外一个选择就是用二分查找树来计算s。这一方法得到的g_0要稍好些：大于或等于$\lfloor\sqrt{x}\rfloor$的最小的2的幂。对某些x的值，这一方法给出更小且足够大的g_0，以便定理的收敛准则仍然成立。这些设计中的差异如下表所示：

205

图11-1中x的范围	图11-2中x的范围	第一个估测值g_0
0	0	0
1	1到3	1
2到4	4到8	2
5到16	9到24	4
17到64	25到80	8
65到256	81到288	16
…	…	…
$2^{28}+1$到2^{30}	$(2^{14}+1)^2$到$(2^{15}+1)^2-1$	2^{15}
$2^{30}+1$到$2^{32}-1$	$(2^{15}+1)^2$到$2^{32}-1$	2^{16}

这一过程如图11-2所示。它特别适合于处理较小的x ($0 \le x \le 24$)，对于这些数不需要做除法。

```
int isqrt(unsigned x) {
   int s, g0, g1;

   if (x <= 4224)
      if (x <= 24)
         if (x <= 3) return (x + 3) >> 2;
         else if (x <= 8) return 2;
         else return (x >> 4) + 3;
      else if (x <= 288)
         if (x <= 80) s = 3; else s = 4;
      else if (x <= 1088) x = 5; else s = 6;
   else if (x <= 1025*1025 - 1)
      if (x <= 257*257 - 1)
         if (x <= 129*129 - 1) s = 7; else s = 8;
      else if (x <= 513*513 - 1) s = 9; else s = 10;
   else if (x <= 4097*4097 - 1)
      if (x <= 2049*2049 - 1) s = 11; else s = 12;
   else if (x <= 16385*16385 - 1)
      if (x <= 8193*8193 - 1) s = 13; else s = 14;
   else if (x <= 32769*32769 - 1) s = 15; else s = 16;
   g0 = 1 << s;                    // g0 = 2**s.

   // Continue as in Figure 11-1.
```

图11-2 整数平方根，利用二分查找求初始估测值

在基本RISC计算机上，图11-1的算法在最坏情况下的执行时间大约是26+(*D*+6)*n*个周期，其中*D*是除法所用的周期，*n*是while循环的执行次数。图11-2的算法在最坏情况下的执行时间大约是27+(*D*+6)*n*个周期，假设分支指令需要1个周期（二者都是）。下面的表列出了两个算法所要执行的循环的平均次数，设*x*的值在指定的区间内均匀分布。

x	图 11-1	图 11-2
0 到 9	0.80	0
0 到 99	1.46	0.83
0 到 999	1.58	1.44
0 到 9999	2.13	2.06
0 到 $2^{32}-1$	2.97	2.97

如果我们假设1个除法需要20个周期，而且*x*的取值范围是0到9999的话，那么两个算法的
206 执行都需要大约81个周期。

2. 二分查找

既然基于牛顿方法的算法在获取第一个估测值时需要使用某种二分查找，那么为什么不用二分查找完成整个计算呢？这一方法从两个边界着手计算，两个边界可能分别被初始化为0和2^{16}。它把边界的中点当作一个估测值。如果这个中点的平方大于参数*x*，那么就令上界等于这个中点。如果这个中点的平方小于参数*x*，那么就令下界等于中点。当上界和下界的差等于1时，停止这个过程，而结果就是下界。

这一方法回避了除法，但是却需要不少乘法，如果0和2^{16}被用作初始边界的话，需要16个乘法。（这一方法在每一次迭代后精度至少增加1位。）图11-3给出了这一过程的一个变形，这一变形所用的边界比0和2^{16}稍有改进。在大多数RISC计算机上，通过比较$b > a$而不是$b-a > 1$来修改*a*和*b*，图11-3所示的过程还可在循环中节省1个周期。

```
int isqrt(unsigned x) {
   unsigned a, b, m;              // Limits and midpoint.

   a = 1;
   b = (x >> 5) + 8;              // See text.
   if (b > 65535) b = 65535;
   do {
      m = (a + b) >> 1;
      if (m*m > x) b = m - 1;
      else         a = m + 1;
   } while (b >= a);
   return a - 1;
}
```

图11-3 整数平方根，简单二分查找

207 每次迭代开始时，谓词$a < \lfloor \sqrt{x} \rfloor + 1$和$b > \lfloor \sqrt{x} \rfloor$都必须为真。初始值*b*应该是一个容易计算且接近$\lfloor \sqrt{x} \rfloor$的量。合理的初始值是*x*、$x \div 4+1$、$x \div 8+2$、$x \div 16+4$、$x \div 32+8$、$x \div 64+16$，等等。对于较小的参数*x*，这个列表的前半部分是较好的初始边界，而对于较大的

参数x来说，这个列表的后半部分是较好的初始边界。（值$x \div 2+1$是可以接受的，但是可能是无用的，因为$x \div 4+1$在所有情况下都比它好或与它相等。）

图11-3所示的过程有7个变形，它们可以通过用$a+1$代替a，或用$b-1$代替b，或把$m=(a+b) \div 2$变成$m=(a+b+1) \div 2$，或把这些替代表达式结合而机械地生成。

图11-3所示过程的执行时间大约是$6+(M+7.5)n$，其中M是乘法的执行时间，而n是被执行的循环的次数。下表给出了循环的平均执行数目，其中x在给定的范围内均匀分布。

x	循环迭代的 平均次数
0 到 9	3.00
0 到 99	3.15
0 到 999	4.68
0 到 9999	7.04
0 到 $2^{32}-1$	16.00

如果我们假设乘法的执行时间是5个周期，而x的范围是0到9999，那么这一算法的执行时间大约是94个周期。最大的执行时间（$n=16$的时候）大约是206个周期。

如果前导0数目指令可用，初始边界可以通过下面的代码设置：

```
b = (1 << (33 - nlz(x))/2) - 1;
a = (b + 3)/2;
```

208

也就是说，$b=2^{(33-\mathrm{nlz}(x))/2}-1$。对于较小的参数$x$来说，这些是非常好的边界（对$0<x<15$需要1次循环迭代），但是对于较大的参数$x$来说，这只是对图11-3所计算的边界的一个中等程度的改进。对于0到9999之间的x，在与上面相同的假设下，迭代次数的平均值大约是5.45，这给出的执行时间大约是74个周期。

3. 硬件算法

还有一个使用移位和减算法的平方根计算方法，这一算法与图9-2描述的硬件除法非常相似。将这一算法嵌入一台32位计算机的硬件时，它使用一个64位寄存器并初始化为32个0位后面跟着参数x。在每次迭代中，这一64位寄存器被左移位2个位，左移位当前结果y（初始值为0）1个位。然后，从这个64位寄存器的左半部分减去$2y+1$。如果减法的结果是非负的，就用这一结果取代64位寄存器的左半部分，并在y上加1（这一加法不需要加法器，因为y的最末位是0）。如果减法的结果是负的，那么64位寄存器和y不变。迭代要执行16次。

这一算法是在1945年给出的 [JVN]。

也许令人吃惊，与 [JVN] 所述的$64 \div 32 => 32$的硬件除法相比，这一算法所需要的执行时间大约是硬件除法的一半，因为这一算法只需大约一半的迭代，而且两个算法的迭代的复杂度基本一样。

为了从软件设计的角度对这一算法编码，最好是避免使用双字移位寄存器，因为完成一次移位大约需要4个指令。图11-4的算法[GLS1]通过移位y和掩码位m向右移位来实现这一操作。这一算法（平均）大约需要149个基本RISC指令。编码中的两个y|m也可以是y+m。

这一算法的操作与小学所学的方法类似。下例展示在一台8位计算机上计算$\lfloor \sqrt{179} \rfloor$的过程。

```
 1011 0011   x0    Initially, x = 179 (0xB3).
- 1          b1
 ------------
 0111 0011   x1    0100 0000   y1
- 101        b2    0010 0000   y2
 ------------
 0010 0011   x2    0011 0000   y2
-  11 01     b3    0001 1000   y3
 ------------
 0010 0011   x3    0001 1000   y3    (不能减)
-   1 1001   b4    0000 1100   y4
 ------------
 0000 1010   x4    0000 1101   y4
```

209 结果是13，余数10留在寄存器x中。

```
int isqrt(unsigned x) {
   unsigned m, y, b;

   m = 0x40000000;
   y = 0;
   while(m != 0) {                    // Do 16 times.
      b = y | m;
      y = y >> 1;
      if (x >= b) {
         x = x - b;
         y = y | m;
      }
      m = m >> 2;
   }
   return y;
}
```

图11-4 整数平方根，硬件算法

可以通过使用包含带符号右移位31个位的常用技巧取消if x>=b的检测。可以证明，b的高阶位总是0（事实上，$b < 5 \times 2^{28}$），从而简化谓词x>=b（参见2.11节）。结果是，可以用下面的语句组取代*if*语句组。

```
t = (int)(x | ~(x - b)) >> 31;    // -1 if x >= b, else 0.
x = x - (b & t);
y = y | (m & t);
```

假设计算机有或非指令，那么上面的代码可以用7个周期来取代平均3个周期，但是，如果这个语境中的条件分支需要超过5个周期，那么这就是值得做的。

某种程度上，似乎硬件算法比软件花费几百个周期去计算整数平方根容易些。为此，对非常小的参数值，我们提供下面的表达式来计算它的平方根。如果我们预期参数很小，那么这些表达式有助于提高上面所给某些算法的速度。

表达式	正确的范围	使用的指令数（完全RISC）
x	0 到 1	0
$x > 0$	0 到 3	1
$(x + 3) \overset{u}{\div} 4$	0 到 3	2
$x \overset{u}{\gg} (x \overset{u}{\div} 2)$	0 到 3	2
$x \overset{u}{\gg} (x > 1)$	0 到 5	2
$(x + 12) \overset{u}{\div} 8$	1 到 8	2
$(x + 15) \overset{u}{\div} 8$	4 到 15	2
$(x > 0) + (x > 3)$	0 到 8	3
$(x > 0) + (x > 3) + (x > 8)$	0 到 15	5

210

啊，难以琢磨的平方根，
它本应容易计算。
但是，我们计算它的最好办法，
就是使用2的幂和牛顿的迭代方法。

11.2 整数的立方根

对于立方根的计算，牛顿方法不是很好。迭代公式有些复杂：

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right)$$

而且，这里也存在选取好的初始值x_0的问题。

然而，有一个与计算平方根的硬件算法类似的硬件算法，这一硬件算法对软件来说不是很糟。算法如图11-5所示。

算法中的三个加1可以用与1的或取代，因为要被递增的值是偶数。即使做这些修改，在硬件上实现它的价值还是值得怀疑，这主要是因为乘法y*(y+1)的存在。

通过使用编译器优化对y的二次项做强度削减可以回避这一乘法。导入一个无符号变量y2来存放y的二次项的值，每当y取新值时相应地更新y2。在y=0之前插入y2=0。在y=2*y之前插入y2=4*y2。改变对b的赋值成b=(3*y2+3*y+1)<<s（并析出因数3）。在y=y+1之前插入y2=y2+2*y+1。这样，最终的程序除了乘以小常量之外没有其他乘法，而乘以小常量的乘法可以用移位和加取代。这一程序有3个加1，可以把它们改成与1的或。这一程序更快，除非你的计算机有只需2个或更少周期的乘法指令。

211

警告：[GLS1]指出，图11-5的代码和它的强度削减派生版不能通过简单的修改就运用于64位计算机。那样，对变量b的赋值可能产生溢出。可以通过从对b的赋值中删除左移位指令，再在赋值之后插入赋值bs=b<<s，并将if(x>=b){x=x-b...改为if(x>=bs&& b==(bs>>s)){x=x-bs...来回避这一问题。

```
int icbrt(unsigned x) {
   int s;
   unsigned y, b;

   s = 30;
   y = 0;
   while(s >= 0) {                    // Do 11 times.
      y = 2*y;
      b = (3*y*(y + 1) + 1) << s;
      s = s - 3;
      if (x >= b) {
         x = x - b;
         y = y + 1;
      }
   }
   return y;
}
```

图11-5 整数立方根，硬件算法

11.3 整数求幂

1. 使用n的二进制分解计算x^n

当n为非负整数时，计算x^n的一个众所周知的方法涉及n的二进制分解。这一技术可运用于形如$x \cdot x \cdot x \cdot \ldots \cdot x$的表达式计算，这里的$\cdot$是任意的可结合操作，例如加、乘（包括矩阵乘法）以及串连接（例如，$('ab')^3$=ababab）。例如，假设我们希望计算$y=x^{13}$。因为用二进制表示时13是1101（也就是说，13=8+4+1），所以

212

$$x^{13} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1$$

因此，可以如下计算x^{13}：

$$t_1 \leftarrow x^2$$
$$t_2 \leftarrow t_1^2$$
$$t_3 \leftarrow t_2^2$$
$$y \leftarrow t_3 \cdot t_2 \cdot x$$

这里需要5个乘法，比反复乘以x所需的12个乘法少得多。

如果已知指数是取非负整数值的变量，那么可以在子例程中使用上面的技术，如图11-6所示。

对于指数$n > 1$，使用这一方法所需要的乘法数目是：

$$\lfloor \log_2 n \rfloor + \text{nbits}(n) - 1$$

这并非总是乘法的最小数目。例如，对于n=27，二进制分解方法计算如下：

$$x^{16} \cdot x^8 \cdot x^2 \cdot x^1$$

这需要7个乘法。然而，下式所示的设计方案只需6个乘法：

$$((x^3)^3)^3$$

在有些指数下二进制分解方法不是最优的，其中最小指数是n=15时（提示：$x^{15}=(x^3)^5$）。

```
int iexp(int x, unsigned n) {
   int p, y;

   y = 1;                          // Initialize result
   p = x;                          // and p.
   while(1) {
      if (n & 1) y = p*y;          // If n is odd, mult by p.
      n = n >> 1;                  // Position next bit of n.
      if (n == 0) return y;        // If no more bits in n.
      p = p*p;                     // Power for next bit of n.
   }
}
```

图11-6 通过n的二进制分解计算x^n 213

也许令人吃惊的是，至今为止还不存在这样的简单方法，对于所有的n，它都能找到计算x^n的最优乘法序列。惟一已知的方法包含大量搜索。这一问题在 [Knu2，4.6.3节] 中有较详细的讨论。

二进制分解方法的一种变形从左到右扫描指数的二进制表示 [Rib,32] 。这一方法类似于二进制转换十进制的从左到右方法。把y初始化为1，再从左到右扫描指数。当遇到0时，计算y的平方。当遇到1时，计算y的平方再乘以x。用这一方法计算$x^{13}=x^{1101_2}$如下：

$$(((1^2\cdot x)^2\cdot x)^2)^2\cdot x$$

这一方法所需要的乘法数目与图11-6的从右到左的方法所需要的（非平凡）乘法数目相同。

2. Fortran中的2^n

IBM XL Fortran编译器把这一函数定义为：

$$\text{pow2}(n)=\begin{cases}2^n, & 0\leqslant n\leqslant 30\\ -2^{31}, & n=31\\ 0, & n<0 \text{ 或 } n\geqslant 32\end{cases}$$

这里假设n和结果都解释为带符号整数。ANSI/ISO Fortran标准要求，当n<0时结果为0。上面对$n\geqslant 31$的定义似乎在它正确的结果模2^{32}内是合理的，而且它与反复做乘法给出的结果一致。

计算2^n的标准方法是把整数1放入一个寄存器，再把它向左移动n个位置。这不能满足Fortran的定义，因为移位量通常被处理为模64或模32（在32位计算机上），这样对较大或负的移位量将给出错误的结果。

如果计算机中有前导0数目指令，可以用如下的4个指令计算pow2(n) [Shep]：

```
x ← nlz(n ≫u 5);        // 若0 ≤ n ≤ 31则x=32，否则x<32
x ← x ≫u 5;             // 若0 ≤ n ≤ 31则x=1，否则x=0
pow2 ← x ≪ n;
```

上面的右移位操作是“逻辑”操作（无符号传播），即使n是一个带符号的量。

214 如果计算机中没有“nlz”指令，上面指令序列中的这一指令可以用2.11节中给出的$x=0$的检测来替代，并把表达式$x \overset{u}{>>} 5$改成$x \overset{u}{>>} 31$。也许更好的方法是注意到谓词$0 < x < 31$与$x \overset{u}{<} 32$的等价性，简化2.11节中给出的$x \overset{u}{<} y$的表达式；它变成$\neg x \ \&\ (x-32)$。这就给出了一个5指令解（如果计算机有与或指令，则是4指令解）。

```
x ← ¬n & (n – 32);        // 当且仅当0 < n < 31时，x<0
x ← x u>> 31;             // 若0 < n < 31则x=1，否则x=0
pow2 ← x << n;
```

11.4 整数对数

“整数对数”函数指的是函数$\lfloor \log_b x \rfloor$，其中x是正整数，b是大于或等于2的整数。通常b=2或10，我们把这两种函数分别记为“ilog2”和“ilog10”。当不特别指出底时，我们把这样的函数记为“ilog”。

为了方便，可以通过定义ilog(0)=－1把定义扩展到x=0[CJS]。这样定义有以下几个理由：

- 通过下面的公式，函数ilog2(x)与前导0数目函数nlz(x)有简单的关系（包含x=0的情况）。因此，如果这两个函数中的一个在硬件或软件上实现了的话，另一个函数也可以很容易地实现。

$$\text{ilog2}(x) = 31 - \text{nlz}(x)$$

- 使用下面的公式很容易计算$\lceil \log(x) \rceil$。对于x=1，这个公式给出ilog(0)=－1。

$$\lceil \log(x) \rceil = \text{ilog}(x-1) + 1$$

- 对于x=1，下面的等式成立（但它对x=0不成立）。

$$\text{ilog2}(x+2) = \text{ilog2}(x) - 1$$

- 这一定义使下面的数学公式仍成立：

$$\lfloor \log_{10} x \rfloor = \lfloor (\log_{10} 2) \log_2 x \rfloor$$

- 这一定义使得ilog(x)的结果成为一个整数的小稠密集合（在32位计算机上，对于ilog2(x)来说，这个集合是－1到31，x是无符号的），这使它可以直接应用于表索引。

215 - ilog(0)=－1可以很自然地从几个计算ilog2(x)和ilog10(x)的算法中得到。

不幸的是，这一定义不是“x的数字数目”的正确定义，对于所有的x，x的数字数目是ilog(x)+1，但x=0时除外。但是，似乎最好的方法是把x=0看成例外。

对于x<0，ilog(x)无定义。为了扩展它的应用范围，我们把这个函数定义为把无符号数映射到带符号数。因此，不能出现负的参数。

1. 以2为底的整数对数

ilog2(x)的计算本质上与计算前导0数目相同，而前导0数目的计算已在5.3节中进行了讨论。对那一节所介绍的算法进行简单修改就可直接用于计算ilog2(x)，无需通过计算nlz(x)并从31中减去结果来计算ilog2(x)。（对于图5-11所示的算法，把`return pop(~x)`改成`return`

pop(x)-1)。

2. 以10为底的整数对数

这一函数可应用于把1个数转换成十进制数，这需要包含1个消除前导0的语句。转换过程不断除以10产生最小有效数字。事先知道最小有效数字的放置位置很有用，这样可以避免将转换的结果存放在临时区域，然后再移动它。

利用表搜索来计算ilog10(x)是一个合理的方法。可以使用二分查找；但是，因为表比较小，而且在大多数的应用中，x通常较小，所以简单的线性搜索也许最好。图11-7给出一个相当简明的程序。

```
int ilog10(unsigned x) {
   int i;
   static unsigned table[11] = {0, 9, 99, 999, 9999,
      99999, 999999, 9999999, 99999999, 999999999,
      0xFFFFFFFF};

   for (i = -1; ; i++) {
      if (x <= table[i+1]) return i;
   }
}
```

图11-7 以10为底的整数对数，简单表搜索

用基本RISC指令，可以把这一程序实现为需要大约9+4 $\lfloor \log_{10} x \rfloor$个指令的代码。因此，它的执行需要5到45个指令，典型的情况（对于$10 < x < 99$）可能使用13个指令。

图11-7所示的程序很容易修改成“寄存器内”版本（不使用表）。这样的程序的可执行部分如图11-8所示。如果计算机有乘以10的快速方法的话，那么这一程序就可能很有用。 216

```
p = 1;
for (i = -1; i <= 8; i++) {
   if (x < p) return i;
   p = 10*p;
}
return i;
```

图11-8 以10为底的整数对数，反复乘以10

在基本RISC计算机上，这个程序需要执行大约10+6 $\lfloor \log_{10} x \rfloor$个指令（将乘当作一个指令）。对于$10 < x < 99$，所需的指令数大约为16个。

可以使用二分查找，给出一个无循环且不使用表的算法。这样的算法可能会把x与10^4比较，再与10^2或10^6比较，等等，直到找到一个指数n使得$10^n < x < 10^{n+1}$。这一路径的执行大约需要10到18个指令，其中4到5个是分支指令（最后的无条件分支计算在内）。

图11-9所示的程序是一个二分查找的改进算法，它在任何一个路径上都有最多4个分支，它更适合于较小的x。对于$10 < x < 99$，它需要6个基本RISC指令，对于$x > 100$，大约需要11到16个指令。

这一程序中的移位指令是带符号移位（这就是使用强制类型转换(int)的原因）。如果计算机中没有这一指令，可以选择下面的一种使用无符号移位的替代方法。这些描述的都是第

一个return语句的替代方法。不幸的是，为了有效地实现，前两个return语句的替代方法
217 使用了立即减，而大多数计算机没有这一指令。最后一条return语句包含一个与较大常量的加（2个指令），但是这对第二和第三个return语句来说不是问题，因为它们无论如何都需要与较大的常量相加。这个较大的常量是$2^{31}-1000$。

```
int ilog10(unsigned x) {
   if (x > 99)
      if (x < 1000000)
         if (x < 10000)
            return 3 + ((int)(x - 1000) >> 31);
         else
            return 5 + ((int)(x - 100000) >> 31);
      else
         if (x < 100000000)
            return 7 + ((int)(x - 10000000) >> 31);
         else
            return 9 + ((int)((x-1000000000)&~x) >> 31);
   else
      if (x > 9) return 1;
      else       return ((int)(x - 1) >> 31);
}
```

图11-9 以10为底的整数对数，改进的二分查找

```
return 3 - ((x - 1000) >> 31);
return 2 + ((999 - x) >> 31);
return 2 + ((x + 2147482648) >> 31);
```

第四个return语句的另一个选择是：

```
return 8 + ((x + 1147483648) | x) >> 31;
```

这里的较大常量是$2^{31}-10^9$。这既回避了与否又回避了带符号移位。

最后的*if-else*结构的其他选择是：

```
return ((int)(x - 1) >> 31) | ((unsigned)(9 - x) >> 31);
return (x > 9) + (x > 0) - 1;
```

其中任何一个都可以节省1个分支。

如果nlz(*x*)或ilog2(*x*)可以当作指令的话，那么有计算ilog10(*x*)的更好更有趣的方法。例如，图11-10的程序用两个查表来计算ilog10(*x*) [CJS]。

由table1可以得到ilog10(*x*)的近似值。这个近似值往往就是正确值，但是对于*x*=0和*x*取范围8~9、64~99、512~999以及8192~9999之间的值，这个近似值比正确值大1。第二个表给出了这样的值，即在一定范围内低于该值的*x*的对数值是近似值减1。

在这一设计中，表用了总共73个字节，使用IBM System/370仅需要6个指令[CJS]。（为了得到这一6指令解，table1中的值必须是这里显示的值的4倍。）在有前导0数目指令而没有其他深奥的指令的RISC计算机上，执行这一算法大约需要10个指令。下面要讨论的其他方法是这一方法的变形。

第一个变形取消由*if*语句生成的条件分支。事实上，如果计算机有设置无符号小于指令的

话，图11-10的程序可以编写成无分支代码，但是下面要讨论的方法可以用于没有不常见指令（除前导0数目指令外）的计算机上。

这一方法把*if*语句替换为在1个减法后面跟着1个右移位31个位的指令，这样，可以从*y*中减去*x*-table2[*y*]的符号位。当*x*较大时（$x \geq 2^{31} + 10^9$）产生了一个困难，这一困难可以通过在table2中添加一个元素来解决，如图11-11所示。 218

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {9, 9, 9, 8, 8, 8,
      7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
      2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[10] = {1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000, 100000000, 1000000000};

   y = table1[nlz(x)];
   if (x < table2[y]) y = y - 1;
   return y;
}
```

图11-10 通过以2为底的对数求以10为底的对数，双表搜索

```
int ilog10(unsigned x) {
   int y;
   static unsigned char table1[33] = {10, 9, 9, 8, 8, 8,
      7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3,
      2, 2, 2, 1, 1, 1, 0, 0, 0, 0};
   static unsigned table2[11] = {1, 10, 100, 1000, 10000,
      100000, 1000000, 10000000, 100000000, 1000000000,
      0};

   y = table1[nlz(x)];
   y = y - ((x - table2[y]) >> 31);
   return y;
}
```

图11-11 通过以2为底的对数求以10为底的对数，双表搜索，无分支

在有前导0数目指令而其他指令却相当基本的RISC计算机上，执行图11-11的算法大约需要11个指令。可以修改这一算法，使得当x=0时返回0而不是-1（这有利于十进制转换问题），修改的方法是把table1中的最后一个条目改成1（即，把“0，0，0，0”改成“0，0，0，1”）。

下一个变形是用1个减法、1个乘法和1个移位代替第一个查表操作。它之所以可行是因为$\log_{10}x$和$\log_2 x$相差一个常数倍，即$\log_{10}2$=0.30103...。因此，对某个适当的$c \approx 0.30103$，通过计算$\lfloor c\,\mathrm{ilog2}(x) \rfloor$，并使用如图11-11中的table2那样的表来修正计算结果，以此来求ilog10(x)。 219

为了说明这一方法，设$\log_{10}2 = c + \varepsilon$，其中$c>0$是$\log_{10}2$的一个有理数近似值，它是一个方便的乘数，而且$\varepsilon>0$。那么对于$x \geq 1$，有

$$\mathrm{ilog10}(x) = \lfloor \log_{10} x \rfloor = \lfloor (c+\varepsilon)\log_2 x \rfloor$$

$$\lfloor c \log_2 x \rfloor \leq \mathrm{ilog10}(x) = \lfloor c \log_2 x + \varepsilon \log_2 x \rfloor$$

$$\lfloor c\,\mathrm{ilog2}(x) \rfloor \leq \mathrm{ilog10}(x) \leq \lfloor c\,(\mathrm{ilog2}(x) + 1) + \varepsilon \log_2 x \rfloor$$

$$\leqslant \lfloor c\ \mathrm{ilog2}(x) + c + \varepsilon \log_2 x \rfloor$$
$$\leqslant \lfloor c\ \mathrm{ilog2}(x) \rfloor + \lfloor c + \varepsilon \log_2 x \rfloor + 1$$

因此，如果我们选择c使得$c+\varepsilon\log_2 x<1$，那么 $\lfloor c\ \mathrm{ilog2}(x) \rfloor$ 逼近ilog10(x)，误差为0或+1。另外，如果我们取ilog2(0)=ilog10(0)=−1，那么 $\lfloor c\ \mathrm{ilog2}(0) \rfloor$ = ilog10(0)（因为$0<c\leqslant 1$），所以我们不必担心这一情况。（在这里也可以把对数定义为ilog2(0)=ilog10(0)=0，等等。）

因为$\varepsilon = \log_{10}2 - c$，故我们必须选择这样的$c$，使得有

$$c + (\log_{10}2 - c)\log_2 x < 1，或$$
$$c(\log_2 x - 1) > (\log_{10}2)\log_2 x - 1$$

当x=1和2时上面的表达式成立（因为$c<1$）。对于更大的x，我们必须有：

$$c > \frac{(\log_{10}2)\log_2 x - 1}{\log_2 x - 1}$$

x越大对c的要求就越严格。对于32位计算机，$x<2^{32}$，所以选择

$$c > \frac{0.30103 \times 32 - 1}{32 - 1} \approx 0.27848$$

就足够了。因为$c<0.30103$（这是由于$\varepsilon>0$），c=9/32=0.28125是一个方便的值。经验表明，比它更粗糙的值如5/16和1/4是不适用的。

这导致图11-12所示的设计方案，这一设计方案从下方开始进行评估，然后通过加1来修正评估值。在有前导0数目指令的RISC计算机上，把乘作为1个指令时，执行这一算法大约需要11个指令。

```
static unsigned table2[10] = {0, 9, 99, 999, 9999,
   99999, 999999, 9999999, 99999999, 999999999};

y = (9*(31 - nlz(x))) >> 5;
if (x > table2[y+1]) y = y + 1;
return y;
```

图11-12 通过以2为底的对数求以10为底的对数，单表搜索

可以把这一设计方案修改成无分支版本，但对于较大的x（$x>2^{31}+10^9$），这同样遇到麻烦，这一麻烦可以用两种方法来解决。一种方法是使用不同的乘数（19/64）和一个稍微扩展的表。这一算法如图11-13所示（在一台有前导0数目指令的RISC计算机上，把乘作为1个指令时，这一算法大约需要11个指令）。

220

另外一个解决方法是：让查表的结果减去x并与x取或，这样对于$x \geqslant 2^{31}$强制地将结果的符号位设置为1，也就是说，把图11-12的第二个可执行行改为

```
y = y + (((table2[y+1] - x) | x) >> 31);
```

如果乘以19的乘法比乘以9的乘法困难得多的话（使用移位和加来实现的话的确如此），这是一个更好的算法。

```
int ilog10(unsigned x) {
   int y;
   static unsigned table2[11] = {0, 9, 99, 999, 9999,
      99999, 999999, 9999999, 99999999, 999999999,
      0xFFFFFFFF};

   y = (19*(31 - nlz(x))) >> 6;
   y = y + ((table2[y+1] - x) >> 31);
   return y;
}
```

图11-13 通过以2为底的对数求以10为底的对数，单表搜索，无分支

对于64位计算机，选择

$$c > \frac{0.30103 \times 64 - 1}{64 - 1} \approx 0.28993$$

就足够了。值19/64=0.296875是方便的，经验表明，比它更粗糙的值是不适用的。这个算法如下（无分支版本）：

```
unsigned table2[20] = {0, 9, 99, 999, 9999, ...,
   99999999999999999};
y = ((19*(63 - nlz(x)) >> 6;
y = y + ((table2[y+1] - x) >> 63;
return y;
```

第12章　数制中的特殊底

本章讨论几种特殊位置计数制。讨论它们只是出于兴趣和好奇，可能没有任何实际应用。我们只讨论整数，但是它们都能扩展到包括小数点后的数字，这通常代表非整数，但并不总是这样。

12.1　以-2为底

以-2为底时，无论是正整数还是负整数都可以表示成没有明确符号或其他不正规的符号，如最大有效位有负权等[Knu3]。使用的数字是0和1，同底为+2时一样；也就是说，用1或0的串表示的值可以理解为:

$$(a_n \ldots a_3 a_2 a_1 a_0) = a_n(-2)^n + \ldots + a_3(-2)^3 + a_2(-2)^2 + a_1(-2) + a_0$$

从这一表达式可以看出，寻找整数的以-2为底（或“负二进制”）表示的过程是持续除以-2，记录余数。除法必须总是给出余数0或1（所要使用的数字）；也就是说，这一除法是一个模除法。例如，下面的设计展示了如何寻找-3的以-2为底的表示。

$$\frac{-3}{-2} = 2 \text{ 余 } 1$$

$$\frac{2}{-2} = -1 \text{ 余 } 0$$

$$\frac{-1}{-2} = 1 \text{ 余 } 1$$

$$\frac{1}{-2} = 0 \text{ 余 } 1$$

因为我们达到了0商，所以停止这一过程（如果持续这一过程的话，余下的商和余数都是0）。现在，向上读取这些余数，我们知道-3的底-2表示是1101。

表12-1的左边给出了从0000到1111的各种组合的以-2为底的解释，右边是从-15到+15的整数的表示。

223

n位字构成的2^n种可能的位组合惟一地表示特定范围内的所有整数，这不是显然的，但是可以用归纳法来证明。归纳假设是n位字表示如下范围内的所有整数:

n为偶数时，$-(2^{n+1}-2)/3$到$(2^n-1)/3$　　(12-1a)

n为奇数时，$(-(2^n-2)/3)$到$((2^{n+1}-1)/3)$　　(12-1b)

首先，假设n为偶数。对于n=2，能够以-2为底表示的整数是10、11、00和01，即

$$-2、-1、0、1$$

这符合上面(12-1a)，而且在此范围内的每一个整数都被且只被表示一次。

224

对于n+1位字，当它的前导位为0时，可以表示符合(12-1a)的所有整数。另外，当前导位是1时，对于符合(12-1a)的所有整数m，它可以表示$m+2^n$。新的表示范围是:

$$2^n - (2^{n+1} - 2)/3 \text{ 到 } 2^n + (2^n - 1)/3$$

或

$$(2^n - 1)/3 + 1 \text{到} (2^{n+2} - 1)/3$$

表12-1 十进制数与以-2为底的数的转换

n (底-2)	*n* (十进制)	*n* (十进制)	*n* (底-2)	*-n* (底-2)
0	0	0	0	0
1	1	1	1	11
10	-2	2	110	10
11	-1	3	111	1101
100	4	4	100	1100
101	5	5	101	1111
110	2	6	11010	1110
111	3	7	11011	1001
1000	-8	8	11000	1000
1001	-7	9	11001	1011
1010	-10	10	11110	1010
1011	-9	11	11111	110101
1100	-4	12	11100	110100
1101	-3	13	11101	110111
1110	-6	14	10010	110110
1111	-5	15	10011	110001

这与(12-1a)给出的范围相邻接，所以对于$n+1$位的字，在范围$-(2^{n+1}-2)/3$到$(2^{n+2}-1)/3$内的所有整数都可以被惟一表示。用$n+1$代替n时，这就是(12-1b)。

类似地可以证明，当n是奇数时，(12-1a)可以从(12-1b)得出，而且在该范围内的所有整数都能惟一表示。

对于加和减运算，如0+1=1及1-1=0这样的通常法则也适用。因为2被写成110，-1被写成11，等等，所以下面的加法法则适用。连同上面显然的法则，这些法则足以完成加法计算。

$$1 + 1 = 110$$
$$11 + 1 = 0$$
$$1 + 1 + 1 = 111$$
$$0 - 1 = 11$$
$$11 - 1 = 10$$

当进行加和减运算时，有时会有两个进位位。进位位要被加到它们所在的栏上，即使是在做减法。将两个进位位都放在左边下一位，（在可能的情况下）并用11+1=0简化会很方便。如果11被进位到含有两个0的栏上，把一个1放入栏中，而另一个1作为进位。下面是几个例子。

加

```
 11  1 11     11
    1  0  1  1  1     19
+ 1  1  0  1  0  1 +(-11)
 ----------------- ------
 0  1  1  0  0  0      8
```

减

```
1 11  1     1
      1  0  1  0  1      21
  - 1  0  1  1  1  0  -(-38)
    ----------------- -----
1  0  0  1  1  1  1      59
```

225

进位只可能是0、1或11。如果高阶位有进位（1或11），就发生溢出。这一情况对加法和减法都适用。因为有三种可能的进位，所以底-2的加法器比2的补码更复杂一些。

有两种对整数取否的方法。与左移位1个位的自身相加（也就是乘以-1），或是用0减去这个整数。没有像2的补码算术那样既简单又方便的“补和加1”规则。在2的补码中，上述规则用于利用加法器构造减法器（由$A+\bar{B}+1$计算$A-B$）。对于以-2为底的情况，似乎没有那么简单的技术。

以-2为底的整数的乘法很简洁。只需使用规则1 × 1=1及0乘0或乘1都等于0，然后再用以-2为底的加法把各栏加起来。

然而，除法就相当复杂了。设计一个合理的硬件除法算法的确是一个挑战，也就是说，这一算法要依赖于反复的减法和移位。为了明确起见，图12-1给出了一个对于8位计算机的除法算法。这一算法做的是模除法（余数非负）。

```
int divbm2(int n, int d) {          // q = n/d in base -2.
   int r, dw, c, q, i;

   r = n;                           // Init. remainder.
   dw = (-128)*d;                   // Position d.
   c = (-43)*d;                     // Init. comparand.
   if (d > 0) c = c + d;
   q = 0;                           // Init. quotient.
   for (i = 7; i >= 0; i--) {
      if (d > 0 ^ (i&1) == 0 ^ r >= c) {
         q = q | (1 << i);          // Set a quotient bit.
         r = r - dw;                // Subtract d shifted.
      }
      dw = dw/(-2);                 // Position d.
      if (d > 0) c = c - 2*d;       // Set comparand for
      else c = c + d;               // next iteration.
      c = c/(-2);
   }
   return q;                        // Return quotient in
                                    // base -2.
                                    // Remainder is r,
}                                   // 0 <= r < |d|.
```

图12-1 以-2为底的除法

尽管这一算法是用C语言写的，而且在二进制2的补码计算机上得到了验证，但是，那是非实质性的——在某种程度上，应该认为它是抽象的算法。输入量n和d以及除q之外的所有内部变量都是无特定表示的数。输出量q是以-2为底解释的位串。 226

还需要做一些解释。如果输入量是以-2为底解释的，那么这一算法就很难表示成可执行

的形式。例如，检测“if(d>0)”将必须检测d的最大有效位是在一个偶数位上。“c=c+d”中的加法将必须是以-2为底的加法。代码将很难理解。由于这一算法的编码方式，应该认为n和d是无特定表示的数。无论使用什么样的编码，代码展示了所需的算术操作。如果数字被编码成以-2为底的形式，那么在实现这一算法的硬件中，乘以-128的乘法是左移位7个位，除以-2的除法是右移位1个位。

例如，代码进行如下的计算：

divbm2(6, 2)=7 （6除以2是111_{-2}）

divbm2(-4, 3)=2 （-4除以3是10_{-2}）

divbm2(-4, -3)=6 （-4除以-3是110_{-2}）

步骤q=q|(1<<i);只表示简单地把q的第i位设置为1。下一条语句r=r-dw表示通过把除数d左移位来减小余数。

这一算法很难详细说明，但是，我们将尝试给出一般的思路。

考虑决定商的第一个位的值，也就是q的第7位。以-2为底，最大有效位为1的8位数的取值范围在-170到-43之间。因此，忽略溢出的可能性，当（且仅当）商在代数上小于或等于-43时，商的第一个位（最大有效位）是1。

因为$n=qd+r$，且正除数$r \leqslant d-1$，对于正除数，当且仅当$n \leqslant -43d+(d-1)$或$n<-43d+d$时，商的第一个位是1。对于负除数，当且仅当$n \geqslant -43d$时，商的第一个位是1（对于模除法，$r \geqslant 0$）。

因此，商的第一个位是1，当且仅当：

$$(d > 0 \ \& \ \neg(n \geqslant -43d + d)) \mid (d < 0 \ \& \ n \geqslant -43d)$$

忽略$d=0$的可能性，上面的表达式可以写成：

$$d>0 \oplus n \geqslant c$$

其中，如果$d>0$则$c=-43d+d$，如果$d<0$则$c=-43d$。

这是决定商的奇数位位置的值的推理方法。对于偶数位，推理方法相反。因此，检测包含项(i&1)==0。（程序中的^表示异或。）

227

在每次循环中，c被设置为除以d之后在第i位上是1的最小整数（最靠近0）。如果当前的余数r大于c，那么把q的第i位设置为1，而且通过在这个位置上减1再乘以除数d来调整r。这里并非真的需要乘法；只需正确地调整d的位置并做减法。

这一算法不太优美。它很难实现，因为有几个加法、减法和比较，而且甚至有在开始就必须做的（乘以常量的）乘法。读者也许希望有一个“统一”的算法，一个不需要检测参数的符号、依赖于结果做不同事情的算法。然而，这样一个统一的算法对于以-2为底（或2的补码算术）来说可能是不存在的。不存在这样的算法的原因是除法本质上是一个非统一的过程。想一想移位和减类型的最简单的算法。这一算法根本就不能移位，但是对于正参数，可以简单地反复从被除数中减去除数，并对所执行的减法数目计数，直到余数小于除数。但是如果被除数是负的（而除数是正的），计算过程则是反复加除数，直到余数是0或是正的，商是得到的计数的负值。除数是负时，计算过程又不一样了。

尽管这样，对于数的带符号量值（signed-magnitude）表示，除法确实是一个统一的过程。对于这样的表示，量值（本身）是正的，所以算法可以简单地减去量值并进行计数，直到余

数为负为止，然后再设商的符号位为计数与参数的异或，而余数的符号位等于被除数的符号位（这给出了普通的截取除法）。

上面给出的算法可以更统一些：如果除数是负的，首先求除数的补，然后执行已简化了的对应于$d>0$的步骤。然后在最后进行修正。对于模除法，修正是对商取负而保留余数不变。这将把某些检测移出循环，但是从总体上看算法仍不够漂亮。

比较通常使用的数的表示与以-2为底的表示，考察在执行四个基本算术操作时计算机硬件是否能统一处理，是有意义的。我们对“统一”没有精确的定义，但是基本上是指不需要根据参数的符号来决定是否要执行某些操作。我们认为，把结果的符号位设置为参数的符号位的异或是一个统一的操作。表12-2总结了四个基本算术操作对于各种数表示法是否统一处理其操作数。

表12-2 在各种编码中统一的操作

	带符号量值	1的补码	2的补码	以-2为底
加	否	是	是	是
减	否	是	是	是
乘	是	否	否	是
除	是	否	否	否

使用“循环进位（end around carry）”技巧可以统一地完成1的补码的加法和减法。对于加法，所有位，包括符号位，都按通常的二进制方法相加，最左侧位（符号位）的进位加到最小有效位上。这一过程总是会停止的。（也就是说，符号位进位的加不会生成另一个符号位的进位。）

228

对于2的补码乘法，只有当取双字积的右半部为结果时，这一项才是“是”。

最后，我们以考察二进制数和以-2为底的数的互换来结束以-2为底的讨论。

为了把以-2为底的数m转换成二进制数，构建这样的字p，p的偶数位（最小有效位是第0位）的值与m的偶数位的值相同，p的奇数位的值为0，即，p是只带m的正权位的字，同样，构建只带m的负权位的字q，使用二进制的减法做p减q。另一个方法可能稍简单些：提取在负权位置上的位，将这些位各向左移动1个位，运用通常的二进制减法，从原来的数字中减去这个数。

为了把二进制数转换成以-2为底的数，提取在奇数位置上的位（权为2^n的位，n为奇数），将这些位向左移位1个位，再用以-2为底的加法把两个数字相加。这里给出两个例子：

把以-2为底的数转换为二进制数

```
    110111  (-13)
  - 1 0 1   (binary subtract)
  ---------
...111110011 (-13)
```

把二进制数转换为以-2为底的数

```
    110111  (55)
  + 1 0 1   (base -2 add)
  ---------
    1001011 (55)
```

在有固定字长的计算机上，如果超出高阶位的进位可以简单地忽略的话，那么这些转换对负数也适用。为了说明，如果字长是6位的话，右边的例子可以认为是把-9从二进制转换成以-2为底的数。

上面把二进制数转换为以-2为底的数的算法在二进制计算机上无法简单地实现，因为它需要做以-2为底的加法。Schroeppel [HAK, item128] 利用更聪明更有用的转换方法克服了这一困难。对于把以-2为底的数转换成二进制数，他的方法是

$$B \leftarrow (N \oplus \mathbf{0b10\ldots1010}) - \mathbf{0b10\ldots1010}$$

为了明白上式的工作原理，设以-2为底的数由四个数字$abcd$组成。那么，直接用二进制
229 （错误地）解释就是$8a+4b+2c+d$。取异或之后，用二进制解释就是$8(1-a)+4b+2(1-c)+d$。减去8+2之后（二进制），是$-8a+4b-2c+d$，这就是它在以-2为底的解释下的值。

Schroeppel的公式可以用于用B来解N，所以这一公式给出了逆方向转换的3指令方法。综合这些结果，对于32位计算机，我们有如下转换成二进制数的公式：

$$B \leftarrow (N \,\&\, \mathbf{0x55555555}) - (N \,\&\, \neg\mathbf{0x55555555})$$
$$B \leftarrow N - ((N \,\&\, \mathbf{0xAAAAAAAA}) \ll 1)$$
$$B \leftarrow (N \oplus \mathbf{0xAAAAAAAA}) - \mathbf{0xAAAAAAAA}$$

而把二进制数转换成以-2为底的数的公式如下：

$$N \leftarrow (B + \mathbf{0xAAAAAAAA}) \oplus \mathbf{0xAAAAAAAA}$$

12.2 以-1+i为底

对于$i=\sqrt{-1}$，通过以$-1+i$为底，所有复整数（带有整实数和整虚数部分的复数）都可以表示成一个没有明确符号或其他不规则符号的单一的数。令人惊讶的是，只用0和1就可表示复数，而且所有整数都有惟一表示。我们对此不做证明或详述，只是对它做简要的描述。

如何表示整数2并不是一件简单的事情[⊖]。然而，可以通过持续地用底除2并记录余数来解决这一问题。上文的“余数”是什么意思呢？如果可能的话，我们希望除以$-1+i$之后余数是0或1（使得数字是0或1）。为了说明这总是可行的，假设我们要用$-1+i$去除任意复数$a+bi$。那么，我们希望找到q和r使得q是复整数，r=0或1，而且有：

$$a + bi = (q_r + q_i i)(-1 + i) + r$$

其中，q_r和q_i分别是q的实数和虚数部分。取上面等式的实数部分和虚数部分并对q求解得：

$$q_r = \frac{b - a + r}{2}, \quad \text{且}$$

$$q_i = \frac{-a - b + r}{2}$$

230

显然，如果a和b都是偶数或都是奇数，那么，取r=0时，q是一个复整数。另外，如果a和b中一个是偶数而另一个是奇数，那么取r=1时，q是一个复整数。

因此，通过下面的设计方案，整数2可以转换成以$-1+i$为底的数。

因为整数2的实数和虚数部分都是偶数，我们知道余数是0，所以可以简单地做除法：

$$\frac{2}{-1+i} = \frac{2(-1-i)}{(-1+i)(-1-i)} = -1-i \text{ 余 } 0$$

⊖ 有兴趣的读者可以挑战这一问题。

因为$-1-i$的实数和虚数部分都是奇数，我们知道余数是0，所以又可以简单地做除法:

$$\frac{-1-i}{-1+i}=\frac{(-1-i)(-1-i)}{(-1+i)(-1-i)}=i\text{ 余 }0$$

因为i的实数和虚数部分分别是偶数和奇数，余数将是1。最简单的办法是一开始就从被除数中减去1。

$$\frac{i-1}{-1+i}=1\text{（余数为1）}$$

因为1的实数和虚数部分是奇数和偶数，下一个余数将是1。从被除数中减去1，得到:

$$\frac{1-1}{-1+i}=0\text{（余数为1）}$$

因为我们得到0商，过程停止，因此，以$-1+i$为底表示2是1100。

表12-3给出了以$-1+i$为底时，从0000到1111的各种位的组合，以及从-15到15的实数的表示。

表12-3 十进制数与以-1+i为底的数间的转换

n (以$-1+i$为底)	*n* (十进制)	*n* (十进制)	*n* (以$-1+i$为底)	*-n* (以$-1+i$为底)
0	0	0	0	0
1	1	1	1	11101
10	$-1+i$	2	1100	11100
11	i	3	1101	10001
100	$-2i$	4	111010000	10000
101	$1-2i$	5	111010001	11001101
110	$-1-i$	6	111011100	11001100
111	$-i$	7	111011101	11000001
1000	$2+2i$	8	111000000	11000000
1001	$3+2i$	9	111000001	11011101
1010	$1+3i$	10	111001100	11011100
1011	$2+3i$	11	111001101	11010001
1100	2	12	100010000	11010000
1101	3	13	100010001	1110100001101
1110	$1+i$	14	100011100	1110100001100
1111	$2+i$	15	100011101	1110100000001

以$-1+i$为底的数的加法法则如下（除某个位为0的显然情况外):

$$1+1=1100$$
$$1+1+1=1101$$
$$1+1+1+1=111010000$$
$$1+1+1+1+1=111010001$$

$$1+1+1+1+1+1 = 111011100$$
$$1+1+1+1+1+1+1 = 111011101$$
$$1+1+1+1+1+1+1+1 = 111000000$$

231

当两个数相加时，在一栏中能够产生的最大进位是6，所以一栏上最大的和是8（111000000）。这就要求相当复杂的加法器。如果想构筑复数算术计算机，那么毫无疑问，最好的方法是让实数部分与虚数部分分开[⊖]，每个部分使用2的补码这样的方法来表示。

12.3 其他底

以$-1-i$为底的数的性质本质上与上面所讨论的以$-1+i$为底的数的性质相同。如果某个位的组合在一个底之下表示数$a+bi$的话，那么它在另外一个底之下表示数$a-bi$。

分别以$1+i$和$1-i$为底时，也可以仅用数字0和1表示所有复整数。这两个底互为共轭复数，同样$-1+i$和$-1-i$也是如此。使用底$1+i$和$1-i$时，某些整数的表示左边有无数个1位的串，类

232 似于用2的补码表示负整数的情况。使用统一的加法和减法法则时，很自然会发生这种情况，就如同2的补码的情况一样。这样的一个整数是2，以上面两个底中的任何一个都表示成…11101100，因此，对这些底，加法法则相当复杂：1+1=…11101100。

将整数的以-2为底的表示的位组成对，就可以得到正整数和负整数的仅使用数字-2、-1、0和1的以4为底的表示。例如，

$$-14_{\text{decimal}} = 110110_{-2} = (-1)(1)(-2)_4 = -1\times4^2 + 1\times4^1 - 2\times4^0$$

类似地，将复整数的以$-1+i$为底的表示的位组成对，我们就可以得到复整数的使用数字0、1、$-1+i$和i的以$-2i$为底的表示。这过于复杂，不能引起人们的兴趣。

“四分虚数（quater-imaginary）”系统（Knu2）是类似的。它使用数字0、1、2和3（无符号）以$2i$为底表示复整数。为了表示某些整数，如带有奇虚数部分的整数，需要小数点及其右侧的一个数字。例如，以$2i$为底时，i被写成10.2。

12.4 最有效的底是什么

假设要构建一台计算机，试图决定以什么为底表示整数。对于寄存器，可用2态（二进制）、3态、4态等等的回路。那么你将使用哪一个呢？

假设b态回路的代价与b成正比。因此，3态回路的代价比2态回路大50%，4态回路是2态回路的两倍，等等。

假设希望一个寄存器能够存放从0到某个最大数M间的所有整数。以b为底，为从0到M的整数编码需要$\lceil \log_b(M+1) \rceil$个数字（例如用十进制表示从0到999 999之间的整数需要$\log_{10}(1\,000\,000) = 6$个数字）。

也许希望寄存器的代价等于所需的数字数目乘以表示每一个数字的代价：

$$c = k\log_b(M+1)\cdot b$$

其中，c是寄存器的代价，k是比例常数。对于给定的M，我们希望找到代价最小的b。

⊖ 这正是1940年贝尔实验室的George Stibitz复数计算器中所采用的方法[Irvine]。

当b的值使得$dc/db=0$时，上面的函数取最小值。因此，我们有

$$\frac{d}{db}(kb\log_b(M+1)) = \frac{d}{db}\left(kb\frac{\ln(M+1)}{\ln b}\right) = k\ln(M+1)\frac{\ln b-1}{(\ln b)^2}$$

当$\ln b=1$（或$b=e$）时，上式为0。 233

这并不是十分令人满意的结果。因为$e\approx 2.718$，最有效的整数底一定是2或3。哪一个更有效呢？以2为底的寄存器的代价与以3为底的寄存器的代价的比是

$$\frac{c(2)}{c(3)} = \frac{k\cdot 2\log_2(M+1)}{k\cdot 3\log_3(M+1)} = \frac{2\ln(M+1)/(\ln 2)}{3\ln(M+1)/(\ln 3)} = \frac{2\ln 3}{3\ln 2}\approx 1.056$$

因此，以2为底时的代价比以3为底时的代价更大，但仅相差一个很小的量。

根据同样的分析，以2为底时的代价比以e为底时的代价稍大，相差比约为1.062。 234

第13章　Gray码

13.1　Gray码

一次只改变n位中的一位是否能循环遍所有n位的2^n个组合呢？答案是肯定的，这就是Gray码定义的性质。也就是说，Gray码是整数的编码，使得一个Gray码整数与它的后继者之间只有1个位不同。这一概念扩展到任意底的表示，例如十进制数，但是这里我们只讨论二进制Gray码。

尽管有很多种二进制Gray码，但是我们只讨论其中的一种："反射二进制Gray码"。在一般文献中，单说"Gray码"指的就是这种编码。我们将展示如何在整数的Gray码表示上做一些基本操作（一般不做证明），而且我们还将展示几个令人惊奇的性质。

反射二进制Gray码的构造如下。从表示整数0和1的位串0和1开始：

0

1

沿上表的底部的水平轴反射上表，把1放在新生成项的左边，而把0放在原来项的左边：

00

01

11

10

这就是n=2时的反射二进制Gray码。为了得到n=3时的反射二进制Gray码，反射上表，并同前面一样在每一个项前放置0或1。

000

001

011

010

110

111

101

100

根据这一构造，通过对n应用归纳法，很容易看到：2^n个位组合在列表中出现而且只出现一次；每一个组合到下一个组合只有1个位发生变化；从最后一个组合循环到第一个组合时，只有1个位发生变化。有最后这条性质的Gray码叫做"循环"Gray码，而且反射二进制Gray码一定是循环Gray码。

如果n>2，存在刚好取遍所有2^n个值的非循环Gray码。例如，000、001、011、010、110、100、101、111就是一组这样的非循环Gray码。

对于n=4，图13-1给出了用普通二进制编码和Gray码编码的整数。公式展示了如何在位对位的级别上把一种表示转换成另一种表示（正如在硬件上实现的那样）。

对于n位Gray码的数目问题，要注意旋转一个循环二进制Gray码或对循环Gray码的各列重新排序后仍是一个循环二进制Gray码。任何一个这样的操作都产生一个不同的编码。因此，在n位上至少有$2^n \cdot n!$种循环二进制Gray码。当$n > 3$时，循环二进制Gray码的数目比这个数目还大。

由图13-1所给出的公式可以看出，Gray码与二进制表示之间有如下的对偶关系：

- Gray码整数的第i位是相应二进制整数中第i位及其左侧1个位的奇偶性（如果第i位的左侧没有其他位则用0代替）。
- 二进制整数的第i位是相应Gray码整数中第i位及其左侧1个位的奇偶性。

二进制码 *abcd*	Gray码 *efgh*		
0000	0000	从二进制码到Gray码	从Gray码到二进制码
0001	0001	$e = a$	$a = e$
0010	0011	$f = a \oplus b$	$b = e \oplus f$
0011	0010	$g = b \oplus c$	$c = e \oplus f \oplus g$
0100	0110	$h = c \oplus d$	$d = e \oplus f \oplus g \oplus h$
0101	0111		
0110	0101		
0111	0100		
1000	1100		
1001	1101		
1010	1111		
1011	1110		
1100	1010		
1101	1011		
1110	1001		
1111	1000		

图13-1 4位Gray码及转换公式

仅用2个指令就可以容易地实现二进制表示到Gray码表示的转换：

$$G \leftarrow B \oplus (B \overset{u}{\gg} 1)$$

把Gray码表示转换成二进制表示更加困难一些，下面的指令给出了一种转换方法：

$$B \leftarrow \bigoplus_{i=0}^{n-1} G \overset{u}{\gg} i$$

我们已经在5.2节的“字的奇偶性计算”中看到过这一公式。正如那里所提到的那样，对于n=32，这一公式可以用下面的指令来计算：

```
B = G ^ (G >> 1);
B = B ^ (B >> 2);
B = B ^ (B >> 4);
B = B ^ (B >> 8);
B = B ^ (B >> 16);
```

因此，一般它需要$2 \cdot \lceil \log_2 n \rceil$个指令。 236

因为很容易把二进制表示转换成Gray码表示，生成连续的Gray码整数是很容易做到的:

```
for (i = 0; i < n; i++) {
   G = i ^ (i >> 1);
   output G;
}
```

13.2 递增Gray码整数

从理论上讲，使用布尔代数记法递增4位二进制整数*abcd*的公式如下:

$$\begin{aligned} d' &= \bar{d} \\ c' &= c \oplus d \\ b' &= b \oplus cd \\ a' &= a \oplus bcd \end{aligned}$$ 237

因此，在硬件上创建一个Gray码计数器的方法就是用上面的逻辑公式创建一个二进制计数器，然后通过相邻位的异或把输出a'、b'、c'、d'转换成Gray码整数，就如图13-1所示的从二进制码到Gray码的转换那样。

下面的公式给出了更好一些的方法:

$$\begin{aligned} p &= e \oplus f \oplus g \oplus h \\ h' &= h \oplus \bar{p} \\ g' &= g \oplus hp \\ f' &= f \oplus g\bar{h}p \\ e' &= e \oplus f\bar{g}\bar{h}p \end{aligned}$$

也就是说，一般的情况是

$$G'_n = G_n \oplus (G_{n-1}\bar{G}_{n-2}\ldots\bar{G}_0 p), \quad n \geqslant 2$$

因为奇偶性p在0与1之间变化，所以一个计数器回路可能会在单独的一位寄存器中保存p，而且在每一次计数时简单地反转p。

在软件方面，寻找Gray编码整数G的后继者G'的最好方法可能就是简单地把G转换成二进制码，递增该二进制字，然后再把它转换回Gray码。另外一个有趣而且几乎同样好的方法就是决定需要反转G中的哪一位。用与G做异或的字的形式表示，反转位的模式如下:

1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16

聪明的读者会发现，上面第i个数正是图13-1中从（二进制）i到$i+1$时最左边的发生变化的位置的掩码。因此，为了递增一个Gray码整数G，把1加到相应的二进制表示时，最左边发生变化的位就是G的位反转位的位置。

由此可得如下递增Gray码整数的算法。这两个算法都通过index(G)首先将G转换成二进制码。

```
B = index(G);                    B = index(G);
B = B + 1;                       M = ~B & (B + 1);
Gp = B ^ (B >> 1);               Gp = G ^ M;
```

238

图13-2 递增Gray码整数

递增Gray码整数的手工计算如下所示:

从右边开始，找到第一个这样的位置，这一位和它左侧的位的奇偶性是偶数。反转这一位。

或，等价地:

设字G的奇偶性是p。如果p是偶数，反转最右侧位。如果p是奇数，反转最右侧1位的左边的位。

后者的规则可以用上面给出的布尔等式直接表示。

13.3 负二进制Gray码

如果以-2为底书写整数，而且使用把（直接）二进制数转换成Gray码的“移位和异或”来转换它们，就得到一个Gray码。3位这样的Gray码具有取遍以-2为底的3位整数的索引，即从-2到5。类似地，对应于以-2为底的4位整数的4位Gray码具有取值范围在-10到5的索引。这不是反射Gray码，但是它几乎是一个反射Gray码。可以通过以下方法生成4位Gray码：从0和1开始，在这一序列的顶部沿水平轴反射这一序列，再在这一序列的底部沿水平轴反射这一序列，以此类推。它是循环的。

为了把这一Gray码转换回以-2为底的整数，转换规则当然与把普通的反射二进制Gray码转换成直接二进制整数的规则相同（因为，无论如何解释位串，这些操作都是可逆的）。

13.4 简史及应用

Gray码是以Frank Gray命名的。Frank Gray是贝尔电话实验室的物理学家，他在20世纪30年代发明了我们现在用于广播彩色电视的方法，这一方法与当时所用的黑白电视的发送和接受方法相容；也就是说，当彩色信号被黑白电视机接受时，图像呈现灰度影像。

Martin Gardner[Gard]论述了Gray码的应用，包括中式圈（九连环）、汉诺塔以及运用表示超立方体的图的哈密尔顿路径问题。他还给出了把整数的十进制表示转换成Gray码表示的方法。

Gray码被运用于位置传感器。一个条形物质被划分成若干传导区和非传导区，分别对应于Gray码整数的1和0。每一列有一个用于决定读取位置的传导线刷。如果传导线刷被置于两个量化位置的分界线处，使得数据的读取不明确时，不考虑解决不明确性的方法。只能有一把不明确的刷子，把它解释成0和1分别给出分界线相邻的一个位置。

239

条状物也可以是一组同心圆的轨道，这给出旋转位置传感器。对于这一应用，Gray码必须是循环的。图13-3给出了一个这样的传感器，其中的4个点表示刷子。

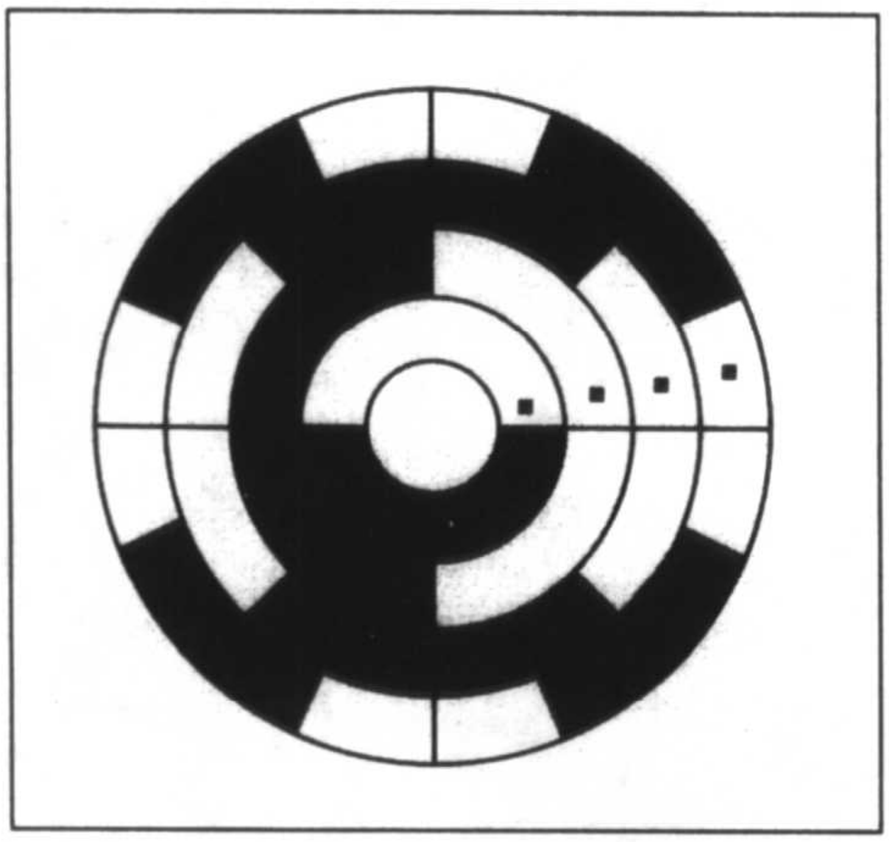


图13-3 旋转位置传感器

240

第14章　Hilbert曲线

1890年，Giuseppe Peano发现了一种平面曲线[⊖]，这一曲线具有相当惊人的性质，即“空间填充性”。这一平面曲线沿单位正方形迂回而行，到达每一个点(x, y)至少一次。

Peano曲线基于将单位正方形的每一个边分成三个相等的部分，这样把这一正方形分成九个小正方形。他的曲线按着特定顺序穿过这九个小正方形。然后，相似地把九个小正方形中的每一个再分成更小的九个小正方形，修改曲线，按一定顺序通过所有这些小正方形。这条曲线可以用以3为底的分数描述，事实上，这就是Peano最初描述它的方式。

1891年，David Hilbert[Hil]发现了Peano曲线的一种变形，这种变形基于把单位正方形的每个边分成两个相等的部分，也就是把单位正方形分成四个小正方形，然后，类似地再把四个小正方形的每一个分成四个小正方形，以此类推。对这种分法的每一个阶段，Hilbert给出了一条穿过所有正方形的曲线。Hilbert曲线——有时称为“Peano-Hilbert曲线”——是这一划分的极限。这一曲线可以用以2为底的分数表示来描述。

图14-1给出了产生Hilbert空间填充曲线的前三步，正如他在1891年的论文中所描绘的那样。

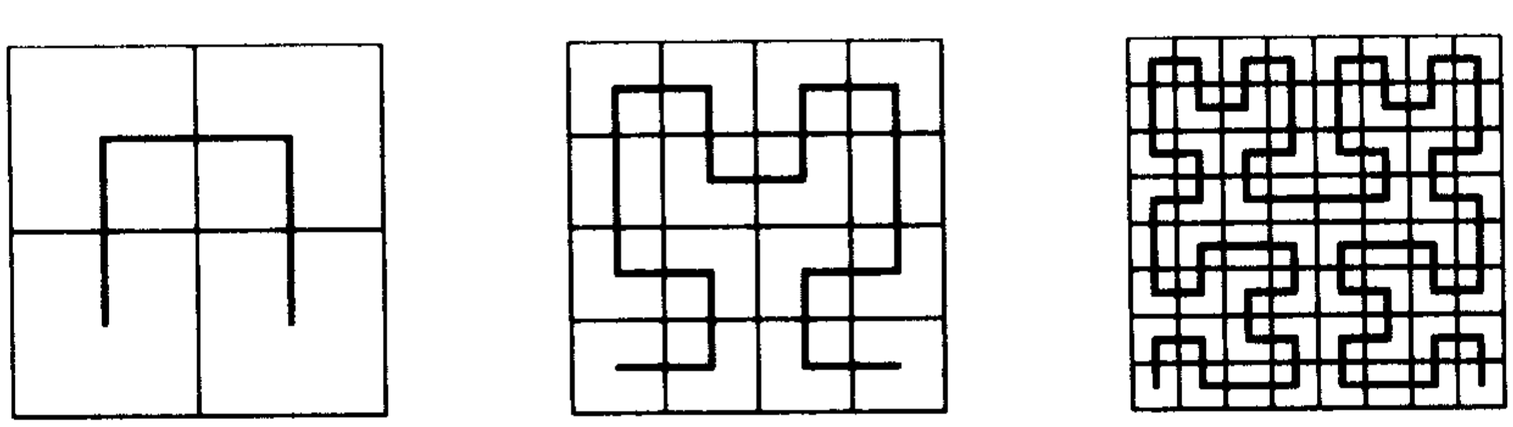


图14-1　定义Hilbert曲线的序列中的前三条曲线

这里，我们采用略微不同的方式。我们把极限为Hilbert空间填充曲线的任何序列中的曲线都称之为“Hilbert曲线”。“n阶Hilbert曲线”意指这样的曲线序列中的第n个曲线。在图14-1中，三条曲线分别是1、2和3阶曲线。我们把这些曲线向下、向左移动，使得曲线的拐角处与上面盒子的交差点重合。最后，我们把n阶曲线放大2^n倍，使得曲线拐角处的坐标是整数。因此，n阶Hilbert曲线的拐角处的两个坐标x和y的取值范围都是从0到2^n-1。我们把曲线坐标值(x, y)从$(0, 0)$到$(2^n-1, 0)$的方向称为正向。下面图14-2给出我们所说的意义下的1阶到6阶“Hilbert曲线”。

14.1　生成Hilbert曲线的递归算法

为了弄清如何生成Hilbert曲线，观察图14-2的曲线。1阶曲线的走向是先向上，再向右，再向下。2阶曲线也是完全遵从这一模式。第一，画一条上行U型曲线，形成一个网。第二，向上画一条单位线段。第三，画一条右行U型曲线，向右画一条单位线段，再画一条右行U型曲线。最后，向下画一条单位线段，再画一条下行U型曲线，构成一个网。

⊖　回忆一下，一条曲线是一个从一维空间到n维空间的连续函数。

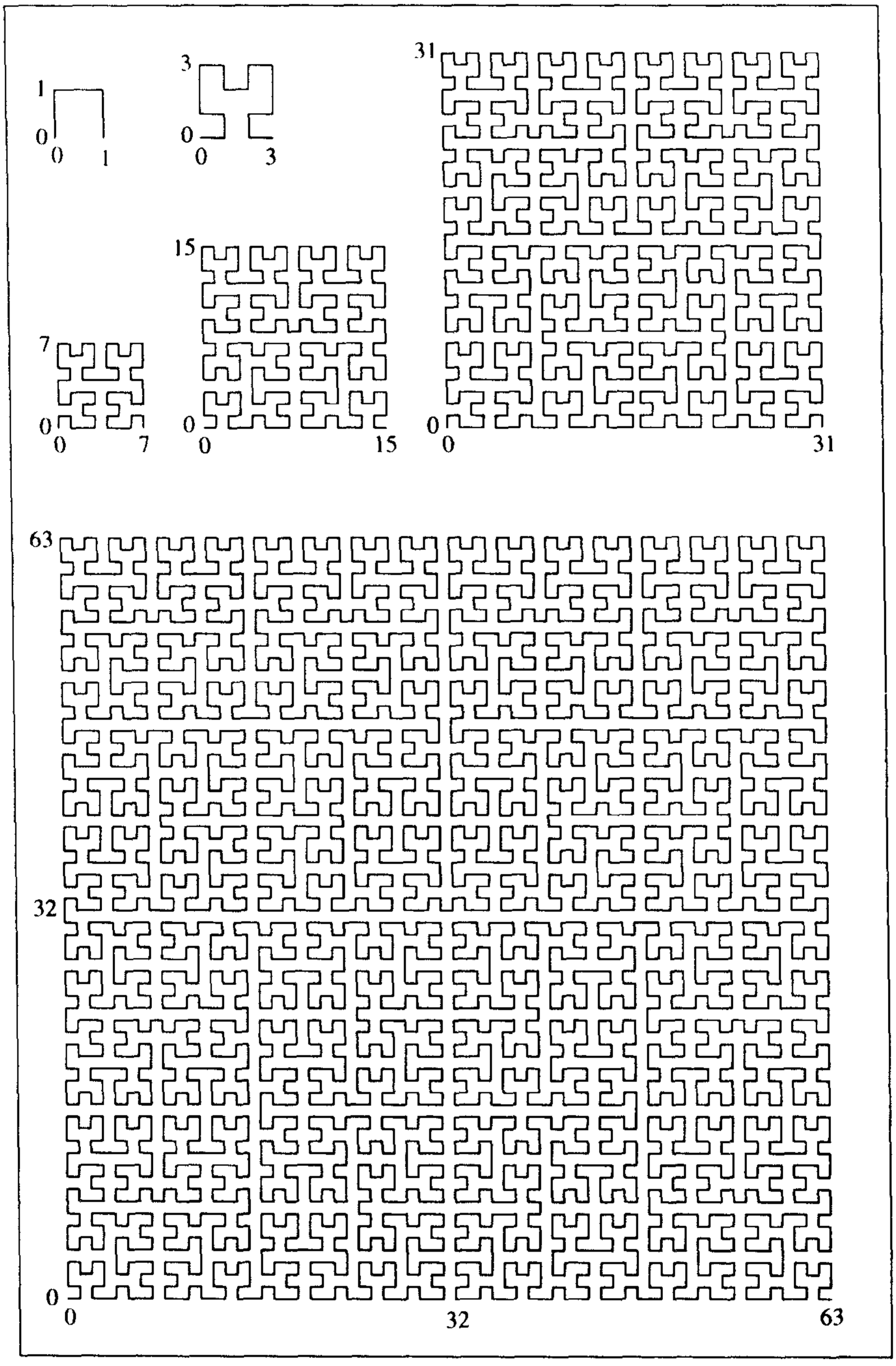


图14-2 1到6阶Hilbert曲线

1阶的倒置U型曲线转换成了2阶的Y型曲线。

我们可以把任意阶的Hilbert曲线看成是一系列不同走向的U型曲线，除了最后一条外，每一个U型曲线都跟着一个特定方向的单位线段。在将某一阶的Hilbert曲线转换成下一阶的Hilbert曲线时，每一个U型曲线被转换成总体走向相同的Y型曲线，每一个单位线段被转换成同样方向的单位线段。

1阶Hilbert曲线（一个按顺时针旋转右行的U型曲线）到2阶Hilbert曲线的转换过程如下:

1) 画一条按逆时针旋转的上行U型曲线。
2) 向上画一条单位线段。
3) 画一条按顺时针旋转的右行U型曲线。
4) 向右画一条单位线段。
5) 画一条按顺时针旋转的右行U型曲线。
6) 向下画一条单位线段。
7) 画一条按逆时针旋转的下行U型曲线。

通过观察，我们可以发现，所有与1阶Hilbert曲线走向相同的U型曲线都能用同样的方法转换得到。可以生成类似的规则来转换其他走向的U型曲线。这些规则被具体体现在图14-3的递归程序中[Voor]。在这一程序中，通过指定了U型曲线的网线方向和旋转方向的两个整数，刻画U曲线的走向，编码如下：

dir = 0: right　　rot = +1: clockwise
dir = 1: up　　rot = −1: counterclockwise
dir = 2: left
dir = 3: down

242
~
243

实际上，dir可以取其他值，但是我们要的是它模4的同余。

```
void step(int);

void hilbert(int dir, int rot, int order) {

   if (order == 0) return;

   dir = dir + rot;
   hilbert(dir, -rot, order - 1);
   step(dir);
   dir = dir - rot;
   hilbert(dir, rot, order - 1);
   step(dir);
   hilbert(dir, rot, order - 1);
   dir = dir - rot;
   step(dir);
   hilbert(dir, -rot, order - 1);
}
```

图14-3 Hilbert曲线生成器

图14-4给出了程序hilbert中使用的驱动程序和step函数。程序的输入是即将构造的Hilbert曲线的阶，输出是一个线段列表，给出每一次运动的方向，沿曲线到区间终点的长度，线段终点的坐标。例如，对于2阶，它给出如下列表：

```
0    0000    00 00
0    0001    01 00
1    0010    01 01
2    0011    00 01
1    0100    00 10
1    0101    00 11
0    0110    01 11
```

```
-1    0111    01 10
 0    1000    10 10
 1    1001    10 11
 0    1010    11 11
-1    1011    11 10
-1    1100    11 01
-2    1101    10 01
-1    1110    10 00
 0    1111    11 00
```

```
#include <stdio.h>
#include <stdlib.h>

int x = -1, y = 0;                    // Global variables.
int i = 0;                            // Dist. along curve.
int blen;                             // Length to print.

void hilbert(int dir, int rot, int order);

void binary(unsigned k, int len, char *s) {
/* Converts the unsigned integer k to binary character
form. Result is string s of length len. */
   int i;

   s[len] = 0;
   for (i = len - 1; i >= 0; i--) {
      if (k & 1) s[i] = '1';
      else       s[i] = '0';
      k = k >> 1;
   }
}
void step(int dir) {
   char ii[33], xx[17], yy[17];

   switch(dir & 3) {
      case 0: x = x + 1; break;
      case 1: y = y + 1; break;
      case 2: x = x - 1; break;
      case 3: y = y - 1; break;
   }
   binary(i, 2*blen, ii);
   binary(x, blen, xx);
   binary(y, blen, yy);
   printf("%5d   %s   %s %s\n", dir, ii, xx, yy);
   i = i + 1;                         // Increment distance.
}
int main(int argc, char *argv[]) {
   int order;

   order = atoi(argv[1]);
   blen = order;
   step(0);                           // Print init. point.
   hilbert(0, 1, order);
   return 0;
}
```

244
~
245

图14-4 Hilbert曲线生成器的驱动程序

14.2 从Hilbert曲线的路长求坐标

为了找到沿n阶Hilbert曲线位于（离出发点）路长为s的点p的坐标(x, y)，我们注意到，$2n$位整数s的两个最大有效位决定它所在的主象限。这是因为，任意阶的Hilbert曲线都要遵从1阶曲线的模式。如果s的两个最大有效位的数字是00，那么它就位于左下象限；如果是01则在左上象限；如果是10则在右上象限；如果是11则在右下象限。因此，s的两个最大有效数字决定n位整数x和y的最大有效位，如下所示：

s的最大有效两位	(x, y)的最大有效位
00	(0, 0)
01	(0, 1)
10	(1, 1)
11	(1, 0)

共有八种可能的U型，但在任意Hilbert曲线上，只可能出现其中的四种。表14-1给出了它们的形态图和从s中的两位数到x和y的相应一位的映射。

表14-1 4种可能的映射

A	B	C	D
00 → (0, 0)	00 → (0, 0)	00 → (1, 1)	00 → (1, 1)
01 → (0, 1)	01 → (1, 0)	01 → (1, 0)	01 → (0, 1)
10 → (1, 1)	10 → (1, 1)	10 → (0, 0)	10 → (0, 0)
11 → (1, 0)	11 → (0, 1)	11 → (0, 1)	11 → (1,0)

观察图14-2可知，在所有情况下，如果p在由映射A(⌜↓)表示的U型上，那么在下一个级别的细节中（对于n大于1，每一个U型都由4个小U型组成的，当n=1时，4个小U型退化成4个点），依赖于p在该U型中的长度（把该U型看成1阶Hilbert曲线时，根据s的相应两位的值，p在其中的长度分别是0、1、2和3），p分别在由映射B、A、A和D表示的U型上。类似地，p在映射B(↰)表示的U型上时，在下一级别的细节中，依赖于p在其中的长度0、1、2和3，p分别在影射A、B、B和C表示的U型上。

这些观察引发出如表14-2所示的状态转移表，其中，状态对应于表14-1所给出的映射。 246

使用这一表时，开始于状态A。需要用前导0填充整数s，使得它的长度等于$2n$，其中n是Hilbert曲线的阶。从左到右成对地扫描s的各位。表14-2的第一行表示，如果当前的状态是A而且当前被扫描的s的两位是00，那么输出（0,0）并进入状态B。接着，扫描前进到s的下两位。类似地，表14-2的第二行表示，如果当前的状态是A，被扫描的两位是01，那么输出(0,1）并停留在状态A。

输出位按从左到右的顺序存储。当到达s的末端时，则定义了n位输出量x和y。

例如，假设n=3且

$$s = 110100$$

表14-2 从s计算(x, y)的状态转移表

若当前状态是	而且s的（右侧）下两位是	那么在(x, y)的后面附加	并且进入状态
A	00	(0, 0)	B
A	01	(0, 1)	A
A	10	(1, 1)	A
A	11	(1, 0)	D
B	00	(0, 0)	A
B	01	(1, 0)	B
B	10	(1, 1)	B
B	11	(0, 1)	C
C	00	(1, 1)	D
C	01	(1, 0)	C
C	10	(0, 0)	C
C	11	(0, 1)	B
D	00	(1, 1)	C
D	01	(0, 1)	D
D	10	(0, 0)	D
D	11	(1, 0)	A

因为这一过程开始于状态A，扫描的初始两位是11，所以过程输出（1,0）并进入状态D（第四行）。接着，在状态D和扫描01的情况下，过程输出（0,1）并停留在状态D。最后，过程输出（1,1）并进入状态C，尽管这个状态已经不重要。

因此，输出是（101，011），也就是说，x=5且y=3。

247 完成这些步骤的C程序如图14-5所示。在这一程序中，分别用0到3的整数表示当前状态是A到D。给变量row的赋值中，当前状态与s的下两位连接，给出了0到15之间的整数，这些整

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {

   int i;
   unsigned state, x, y, row;

   state = 0;                                  // Initialize.
   x = y = 0;

   for (i = 2*n - 2; i >= 0; i -= 2) {    // Do n times.
      row = 4*state | (s >> i) & 3;           // Row in table.
      x = (x << 1) | (0x936C >> row) & 1;
      y = (y << 1) | (0x39C6 >> row) & 1;
      state = (0x3E6B94C1 >> 2*row) & 3; // New state.
   }
   *xp = x;                                    // Pass back
   *yp = y;                                    // results.
}
```

图14-5 从s计算(x, y)的程序

数是表14-2中适用的行数。变量row用来访问被用做位串来表示表14-2最右侧两列的整数（用十六进制表示）；也就是说，这些访问是寄存器内查表的操作。程序中的十六进制值从左到右的各位与表14-2的从底到顶的各行对应。

[L&S]给出了一个完全不同的算法。不同于图14-5所给出的算法，它从右到左扫描*s*的各位。这一算法基于如下的观察，我们可以根据1阶Hilbert曲线把*s*的最小两个有效位映射到(*x*, *y*)，再检测*s*的下面两位。如果它们是00，互换刚刚计算出的*x*和*y*的值，这相当于对于直线*x*=*y*反射1阶Hilbert曲线（参见图14-1给出的1阶和2阶Hilbert曲线）。如果这两位是01或10，就不交换*x*和*y*的值。如果是11则交换*x*和*y*，并对它们求补。对于*s*左侧的其他位组按同样的规则处理。这就是表14-3和图14-6的算法。有些好奇的是，首先把*s*的相应两位分别附加到*x*和*y*的前面，再做交换和求补操作，包括最近在前面附加的位；结果不变。

在图14-6中，变量x和y没有被初始化，这可能让某些编译器给出出错信息。但是无论是否初始化x和y，算法都是正确的。

通过2.19节给出的“三异或”技巧完成交换操作，可以避免图14-6所示循环中的分支。 248

表14-3 从s计算(x, y)的Lam and Shapiro法

若*s*的（左侧）下两位是	那么	并在(*x*,*y*)的前面附加
00	交换*x*和*y*	(0,0)
01	不变	(0,1)
10	不变	(1,1)
11	交换*x*和*y*并对它们取补	(1,0)

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {

   int i, sa, sb;
   unsigned x, y, temp;

   for (i = 0; i < 2*n; i += 2) {
      sa = (s >> (i+1)) & 1;        // Get bit i+1 of s.
      sb = (s >> i) & 1;            // Get bit i of s.

      if ((sa ^ sb) == 0) {         // If sa,sb = 00 or 11,
         temp = x;                  // swap x and y,
         x = y^(-sa);               // and if sa = 1,
         y = temp^(-sa);            // complement them.
      }
      x = (x >> 1) | (sa << 31);  // Prepend sa to x and
      y = (y >> 1) | ((sa ^ sb) << 31); // (sa^sb) to y.
   }
   *xp = x >> (32 - n);             // Right-adjust x and y
   *yp = y >> (32 - n);             // and return them to
}                                   // the caller.
```

图14-6 从*s*计算(*x*, *y*)的L&S（Lam and Shapiro）方法

可以用下面的代码替换代码中的*if*语句块，其中swap和cmpl是无符号整数：

```
swap = (sa ^ sb) - 1;   // -1 if should swap, else 0.
cmpl = -(sa & sb);      // -1 if should compl't, else 0.
x = x ^ y;
y = y ^ (x & swap) ^ cmpl;
x = x ^ y;
```

然而，相对于*if*语句块所需要的2个或6个指令，这一代码需要9个指令，所以只有当分支的代价相当高时，这才是一个好的选择。

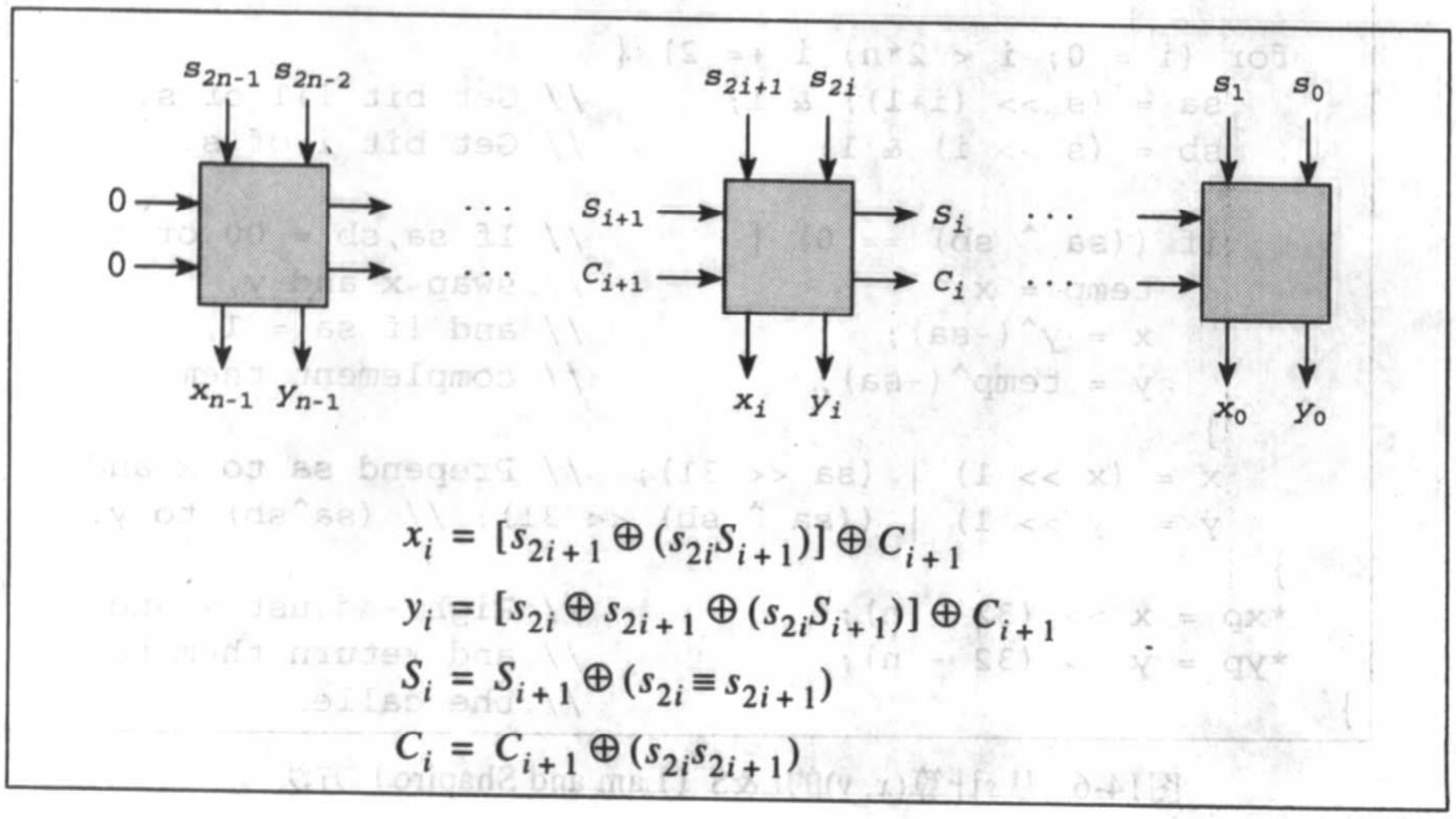
249 [L&S]的“交换和求补”的思路指出了生成Hilbert曲线的逻辑回路。下面描述的回路背后的思路是：当沿n阶曲线行走时，本质上是根据表14-1的映射A把s的位对映射到坐标(x, y)。然而，当行走到各个区域时，映射的输出需要交换、求补，或交换并求补。图14-7记录了每一阶段所需要的交换和求补的要求，它使用适当的映射把s的两个位映射到(x_i, y_i)并生成下一步所需要的交换和求补的信号。

假设有存放路长s的寄存器和递增路长的回路。那么，为了找到Hilbert曲线上的下一个点，首先递增s，再按表14-4中所给的方法对它进行变换。这是一个从左到右的过程，而且稍稍有些问题，因为递增s是一个从右到左的过程。因此，在n阶Hilbert曲线上生成一个新点所需的时间与$2n+n$成比例（$2n$用于递增s，n用于把s转换到(x, y)）。

表14-4 从s 计算(x, y)的逻辑

若s的（右侧）下两位是	那么在(x,y)的后面附加	并设置
00	(0, 0)*	swap = $\overline{\text{swap}}$
01	(0, 1)*	不变
10	(1, 1)*	不变
11	(1, 0)*	swap = $\overline{\text{swap}}$, cmpl = $\overline{\text{cmpl}}$
* 可能是经交换和/或取补后的坐标。		

图14-7以逻辑回路的形式给出了这种计算。在这一图中，S代表交换信号，C代表取补信号。



250 图14-7 通过沿着Hilbert曲线前进一步来递增(x, y)的逻辑回路

图14-7的逻辑回路指出了另一个从s计算(x, y)的方法。注意交换信号和取补信号是如何通过n个阶段从左向右传递的。这表明可以使用并行前缀操作快速地（使用$\log_2 n$步而不是$n-1$步）把交换和取补信息传到每一个阶段，然后再利用图14-7的等式做某些字并行逻辑操作来计算x和y。x和y的值被混放在一个字的偶数位和奇数位上，所以需要通过逆混洗操作将它们分离（参见7.2节）。这看起来似乎比较麻烦，似乎只对相当大的n才值得，但是让我们看一看它是如何进行的。

图14-8给出这一操作的过程[GLS]。这一过程对全字量进行操作，它首先在s的左侧用'01'位填充s。这种位结合对交换和取补没有影响。下一步，计算量cs（求补－交换）。这个字的形式是cscs...cs，它是根据表14-3而来的，其中对于每个c（一位），如果它是1则意味着要对相应的位对求补，每一个值为1的s表示相应的位对要做交换。换句话说，这两个语

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                      unsigned *yp) {
   unsigned comp, swap, cs, t, sr;

   s = s | (0x55555555 << 2*n); // Pad s on left with 01
   sr = (s >> 1) & 0x55555555;  // (no change) groups.
   cs = ((s & 0x55555555) + sr) // Compute complement &
        ^ 0x55555555;           // swap info in two-bit
                                // groups.
   // Parallel prefix xor op to propagate both complement
   // and swap info together from left to right (there is
   // no step "cs ^= cs >> 1", so in effect it computes
   // two independent parallel prefix operations on two
   // interleaved sets of sixteen bits).

   cs = cs ^ (cs >> 2);
   cs = cs ^ (cs >> 4);
   cs = cs ^ (cs >> 8);
   cs = cs ^ (cs >> 16);
   swap = cs & 0x55555555;      // Separate the swap and
   comp = (cs >> 1) & 0x55555555;  // complement bits.

   t = (s & swap) ^ comp;       // Calculate x and y in
   s = s ^ sr ^ t ^ (t << 1);   // the odd & even bit
                                // positions, resp.
   s = s & ((1 << 2*n) - 1);    // Clear out any junk
                                // on the left (unpad).

   // Now "unshuffle" to separate the x and y bits.

   t = (s ^ (s >> 1)) & 0x22222222; s = s ^ t ^ (t << 1);
   t = (s ^ (s >> 2)) & 0x0C0C0C0C; s = s ^ t ^ (t << 2);
   t = (s ^ (s >> 4)) & 0x00F000F0; s = s ^ t ^ (t << 4);
   t = (s ^ (s >> 8)) & 0x0000FF00; s = s ^ t ^ (t << 8);

   *xp = s >> 16;                  // Assign the two halves
   *yp = s & 0xFFFF;               // of t to x and y.
}
```

图14-8 从s计算(x, y)的并行前缀方法

句映射s的每一个位对如下：

s_{2i+1} s_{2i}	cs
0 0	01
0 1	00
1 0	00
1 1	11

这是我们要运用并行前缀操作的量。PP-XOR就是我们要使用的从左到右的并行前缀操作，因为要求补或交换的相继1位与异或具有相同的逻辑性质：两个相继的1位互相抵消。

用同一个PP-XOR操作传播两种信号（求补和交换），每一个作用于cs的相隔位。

下面的四个赋值语句对s的每一个位对到(x,y)的值的转换产生影响，x在奇数位（最左边）上，y在偶数位上。尽管这一逻辑似乎比较模糊，但是很容易验证：s的每一个位对都是根据图14-7前两个布尔等式的逻辑来转换的。（建议：利用t和sr的奇数位上都是0的事实，分别考虑偶数位和奇数位是如何转换的。）

这一过程的剩余部分无需另加说明。相对于图14-6所需的（平均）大约19n+10个指令，这一过程需要66个基本RISC指令（常量，无分支）。（图14-6所需的指令数目基于编译后的代码，包括序言和结尾，这些本质上是空的。）因此，对于$n \geq 3$，并行前缀的方法更快些。

251

14.3 Hilbert曲线上坐标到路长的转换

给定Hilbert曲线上某个点的坐标，从原点到这一点的路长可以通过类似于表14-2的状态转换表来计算。表14-5就是这样的状态转换表。

表14-5 从(x, y)计算s的状态转换表

若当前状态是	而且(x, y)的（右侧）下两个位是	那么在s的后面附加	并且进入状态
A	(0, 0)	00	B
A	(0, 1)	01	A
A	(1, 0)	11	D
A	(1, 1)	10	A
B	(0, 0)	00	A
B	(0, 1)	11	C
B	(1, 0)	01	B
B	(1, 1)	10	B
C	(0, 0)	10	C
C	(0, 1)	11	B
C	(1, 0)	01	C
C	(1, 1)	00	D
D	(0, 0)	10	D
D	(0, 1)	01	D
D	(1, 0)	11	A
D	(1, 1)	00	C

这一方法的解释类似于前一节中的说明。首先，用前导0填充x和y，使得它们的长度是n位，这里的n是Hilbert曲线的阶。其次，从左到右扫描x和y的各位并从左到右构建s。 252

完成这些步骤的C语言程序如图14-9所示。

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i;
   unsigned state, s, row;

   state = 0;                                  // Initialize.
   s = 0;

   for (i = n - 1; i >= 0; i--) {
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      s = (s << 2) | (0x361E9CB4 >> 2*row) & 3;
      state = (0x8FE65831 >> 2*row) & 3;
   }
   return s;
}
```

图14-9 从(x, y)计算s的程序

[L&S]给出了根据(x, y)计算s的算法，这一算法类似从s计算(x, y)的算法（参见表14-3）。这是一个从左到右的算法，如表14-6和图14-10所示。 253

表14-6 从(x, y)计算s的L&S方法

若(x, y)的（右侧）下两个位是	那么	并在s的后面附加
(0,0)	交换x和y	00
(0,1)	不变	01
(1,0)	交换x和y并对它们取补	11
(1,1)	不变	10

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i, xi, yi;
   unsigned s, temp;

   s = 0;                                  // Initialize.
   for (i = n - 1; i >= 0; i--) {
      xi = (x >> i) & 1;                   // Get bit i of x.
      yi = (y >> i) & 1;                   // Get bit i of y.

      if (yi == 0) {
         temp = x;                         // Swap x and y and,
         x = y^(-xi);                      // if xi = 1,
         y = temp^(-xi);                   // complement them.
      }
      s = 4*s + 2*xi + (xi^yi);            // Append two bits to s.
   }
   return s;
}
```

图14-10 从(x, y)计算s的L&S方法

14.4 递增Hilbert曲线上点的坐标

给定n阶Hilbert曲线上某个点的坐标(x, y)，如何寻找下一个点的坐标呢？一种方法就是把(x, y)转换到s，把s加1，然后使用上面给出的算法，再把s的新值转换回(x, y)。

一个稍稍好一些（但并非非常好）的方法是基于这样的事实：当沿着Hilbert曲线移动时，在每一步，或者是x或者是y（而不是两者），要么是递增要么是递减1。下面的算法从左到右扫描坐标来决定最右边两个位所在的U型曲线的类型。然后，根据U型曲线和最右边的两个位递增或递减x或y。

这只是基本情况，但是，当路径是在一条U型曲线（每四步就发生一次）的末端时，问题
254 变得复杂了。在这一点，前进的方向要由x和y的前边的位以及与这些位相关联的更高阶的U型曲线决定。如果这个点也在它的（更高阶的）U型曲线的末端的话，那么x和y的再前一位以及那里的U型曲线决定前进的方向，以此类推。

表14-7描述了这一算法。在这个表中，A、B、C和D代表表14-1所示的U型曲线。为了使用这个表，首先用前导0填充x和y使得它们的长度为n，这里n是Hilbert曲线的阶。从状态A开始、从左到右扫描x和y的各位。表14-7的第一行表示，如果当前的状态是A，并且当前扫描的位是(0,0)的话，那么设一个变量，令它表示要增加y，并进入状态B。其他行的意思类似，负号后缀表示递减相关的坐标。第三列中的破折号表示不改变记录坐标变化的变量。

表14-7 Hilbert曲线上的一步操作

若当前状态是	而且(x,y)的（右侧）下两个位是	那么准备递增/递减	并且进入状态
A	(0, 0)	y+	B
A	(0, 1)	x+	A
A	(1, 0)	—	D
A	(1, 1)	y–	A
B	(0, 0)	x+	A
B	(0, 1)	—	C
B	(1, 0)	y+	B
B	(1, 1)	x–	B
C	(0, 0)	y+	C
C	(0, 1)	—	B
C	(1, 0)	x–	C
C	(1, 1)	y–	D
D	(0, 0)	x+	D
D	(0, 1)	y–	D
D	(1, 0)	—	A
D	(1, 1)	x–	C

扫描x和y的最右边的位之后，变量的最终值指明需要递增或递减的坐标。

实现这些步骤的C语言程序如图14-11所示。变量dx以这样的方式初始化：如果它被多次

调用，那么算法产生循环，一再生成相同的Hilbert曲线。（然而，连结两个循环的步骤不是单位步骤。）

255

```
void hil_inc_xy(unsigned *xp, unsigned *yp, int n) {

   int i;
   unsigned x, y, state, dx, dy, row, dochange;

   x = *xp;
   y = *yp;
   state = 0;                          // Initialize.
   dx = -((1 << n) - 1);               // Init. -(2**n - 1).
   dy = 0;

   for (i = n-1; i >= 0; i--) {                  // Do n times.
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      dochange = (0xBDDB >> row) & 1;
      if (dochange) {
         dx = ((0x16451659 >> 2*row) & 3) - 1;
         dy = ((0x51166516 >> 2*row) & 3) - 1;
      }
      state = (0x8FE65831 >> 2*row) & 3;
   }
   *xp = *xp + dx;
   *yp = *yp + dy;
}
```

图14-11　Hilbert曲线上的一步操作程序

从逻辑上看，表14-7很容易实现，如图14-12所示。在该图中，变量有如下的意义：

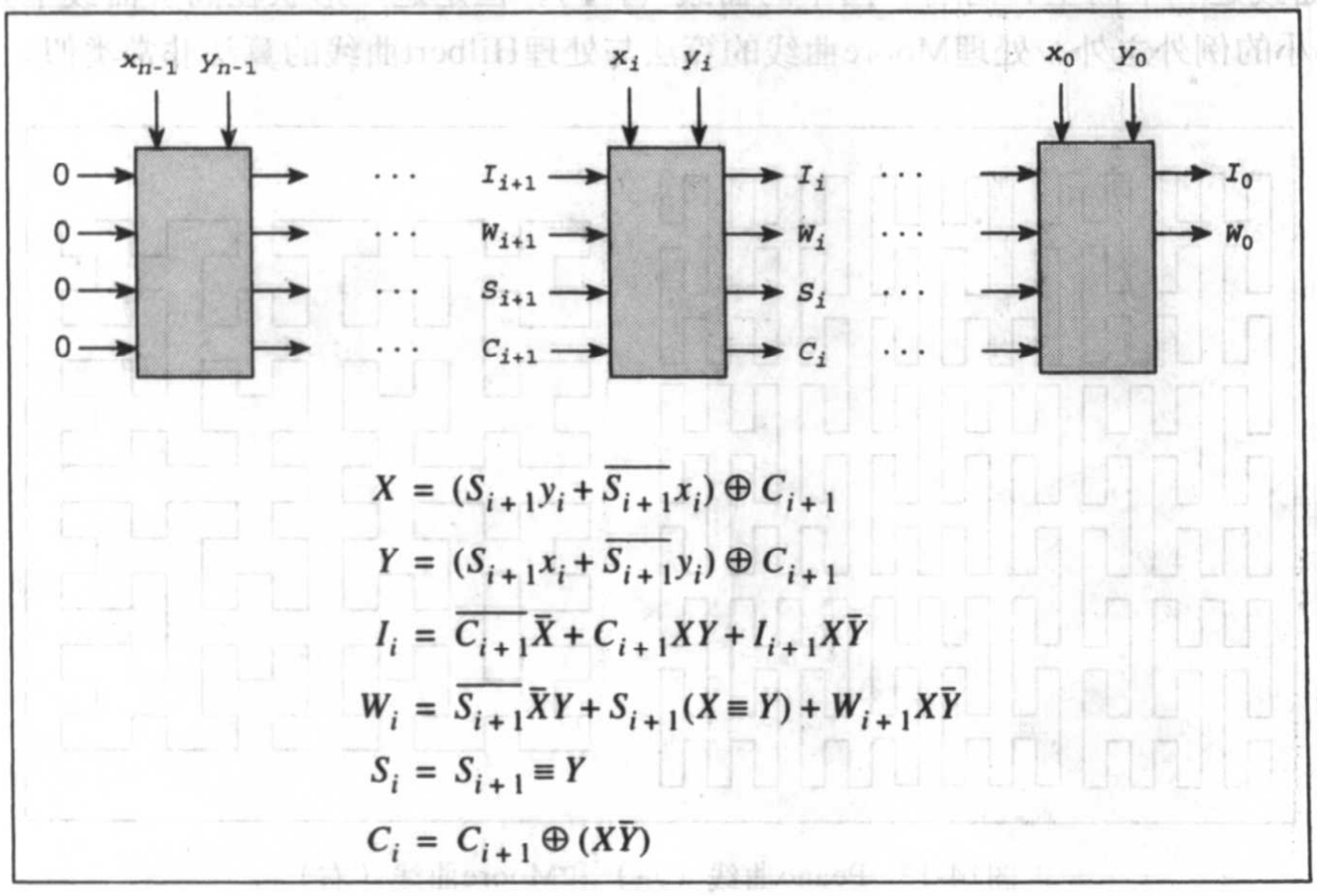


图14-12　Hilbert曲线上的一步操作中的递增(*x*,*y*)的逻辑回路

x_i：输入x的第i位。

y_i：输入y的第i位。

X,Y：根据S_{i+1}和C_{i+1}而交换和填充的x_i和y_i。

I：如果是1则递增，如果是0则递减（1）。

W：如果是1则递增或递减x，如果是0则递增或递减y。

S：如果是1则交换x_i和y_i。

C：如果是1则对x_i和y_i求补。

S和C一起识别表14-7的“状态”，即，(C,S)=(0,0)、(0,1)、(1,0)和(1,1)分别代表状态A、B、C和D。输出信号是I_0和W_0，它们分别告知是递增还是递减，需要改变哪个变量。（除了所表明的逻辑之外，还需要一个递增器/递减器回路，它带有一个传送x或y到递增器/递减器的MUX，
256 并把改变了的值传送回存放x或y的寄存器。另外，也可以用两个递增器/递减器回路。）

14.5 非递归生成算法

表14-2和14-7的算法给出了两个生成任意阶Hilbert曲线的非递归算法。在硬件上实现这两个算法中的任何一个都不太困难。基于表14-2的硬件包含一个存放s的寄存器，在每一步，算法递增s，然后转换成坐标(x,y)。基于表14-7的硬件不需要存放s的寄存器，但是算法更复杂。

14.6 其他空间填充曲线

正如所提到的那样，Peano于1890年首次发现了空间填充曲线。从那以后，人们发现了许多这一曲线的变形，通常都叫做“Peano曲线”。1900年，Eliakim Hastings Moore发现了Hilbert曲线的一个有趣的变形。在某种意义上，这一曲线是一个“循环”曲线，它的末端与开始端相差一步。图14-13给出了3阶的Peano曲线和4阶的Moore曲线。Moore曲线具有不规律性，1阶曲线是一个向上、向右、向下的曲线（⊓），但是这一形状在高阶曲线中并不出现。除了这小小的例外之外，处理Moore曲线的算法与处理Hilbert曲线的算法非常类似。

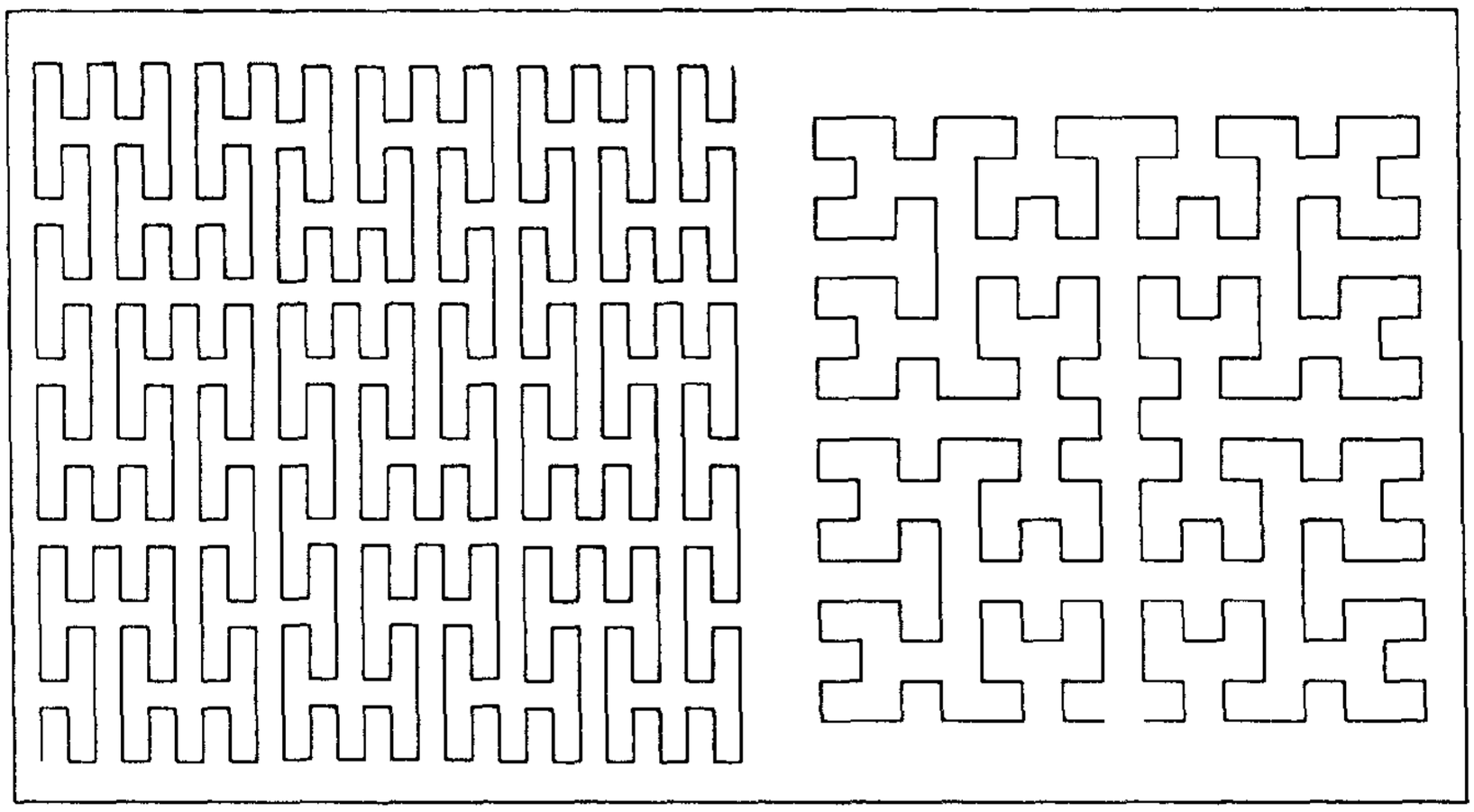
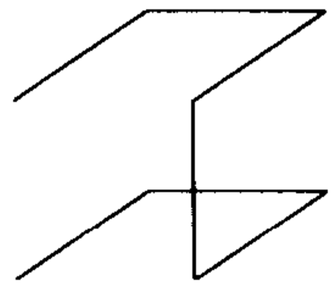


图14-13 Peano曲线（左）和Moore曲线（右）

257 Hilbert曲线可以推广到任意长方形、三维或更高维空间。三维Hilbert曲线的基本构建块

如下所示。它连结2 × 2 × 2立方体的所有八个点。[Sagan]中论述了这些空间填充曲线和其他的空间填充曲线。



14.7 应用

空间填充曲线可应用于图像处理：压缩、半色调和纹理分析[L&S]。还有一个应用是用于改进光线跟踪和图形绘制技术的计算机性能。通常是通过按普通光栅线扫描依次将光线投射到景物来扫描景物（从左到右，然后从上到下扫描屏幕）。当光线碰到模拟景物数据库中的某个物体时，判定这一物体在这个点的颜色和其他特性，并使用这些结果通过送出的光线照亮这一像素。（这一描述有些过度简化，但是对于我们的目的而言已经足够了。）问题是数据库通常很大，随着扫描光线碰到各种物体，必须对每个物体的数据进行页面调入和调出操作。当光线扫描一行时，它常常会碰到前一次扫描时碰到过的物体，这就需要重新调入页面。如果扫描操作具有某种形式的局部特性的话，就可以减少调页操作。例如，扫描完屏幕的一个象限之后再处理另一个象限可能会有帮助。 258

Hilbert曲线似乎具有我们所寻求的局部性质。它递归地完全扫描一个象限之后再开始下一个象限的扫描，而且从一个象限到下一个象限不需要长的跳转（long jump）。

Douglas Voorhies[Voor]对惯常的单一方向的行扫描以及Peano曲线扫描和Hilbert曲线扫描的调页行为进行了模拟试验。他的方法是，把给定尺寸的圆随机地散落在屏幕上。碰到了一个圆的扫描表示碰到了一个新物体，并对这一物体的数据进行页面调入操作。但是，当扫描离开一个圆时，让物体的数据存留在内存中，直到扫描离开半径是物体圆半径的两倍的一个圆为止。因此，如果扫描只是短距离地离开这一物体，然后又回到这一物体，那么就不做调页操作。他在模拟的1024 × 1024的屏幕上使用了很多不同大小的圆重复进行这一实验。

假设进入一个物体圆而且离开包围这一物体圆的圆表示一次调页操作的话，那么显然，普通的行扫描在扫描一个直径为D个像素的圆时，会进行D次调页操作，因为每次进入这个圆的行扫描都离开它的外圆。Voorhies模拟的有趣结论是，对于Peano曲线，扫描一个圆时的调页操作数目大约是2.7，而且，也许令人惊讶的是，这一值与圆的直径无关。对于Hilbert曲线，这个数字大约是1.4，同样与圆的直径无关。因此，经验表明，在减少调页操作上，Hilbert曲线优于Peano曲线，更是远优于普通的行扫描。（调页计数与圆的直径无关这一结果可能是由于外圆直径与物体圆直径成比例这一人为因素造成的。） 259

第15章　浮　　点

上帝创造了整数，
其他都是人的工作。

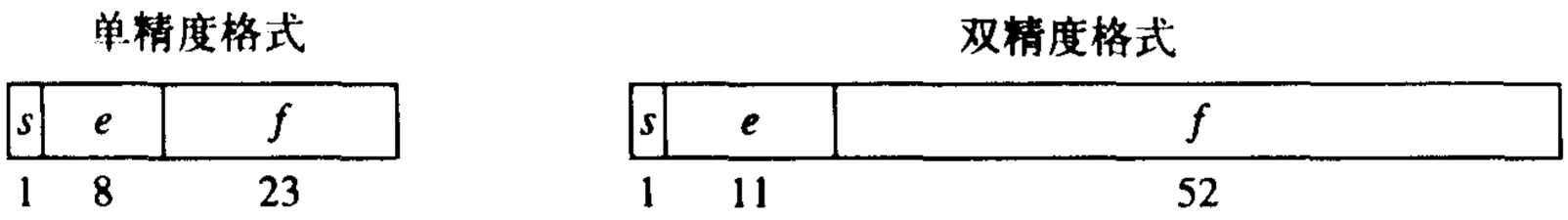
——Leopold Kronecker

用整数算术和逻辑指令对浮点数进行操作通常是一件复杂的事情。对于IEEE二进制浮点算术标准的规则和格式来说尤其如此。IEEE二进制浮点算术标准（IEEE Std.754-1985）通常称“IEEE算术”。IEEE算术有NaN（非数）和无穷，对于几乎所有操作来说，这些都是特殊情况。它有正0和负0，它们之间的比较结果必须是相等的。它还有第四个“不可排序”的比较结果。“规格化”数不明确存储小数的最大有效位，但是“非规格化”数或“亚规格化”数则存储小数的最大有效位。小数以符号真的形式出现，而且指数以偏移的形式出现，而整数几乎普遍是采用2的补码形式。当然，这些情况的存在是有原因的，但是这将导致程序充满比较和分支，对有效实现是一个挑战。

我们假设读者对IEEE标准有所了解，因此在这里只简单加以概述。

15.1　IEEE格式

我们将把注意力限定在IEEE 754所阐述的单精度和双精度（32位及64位）格式上。这一标准还描述了“扩展单精度”和“扩展双精度”格式，但是对这些只做了不精确的描述，因为细节依赖于实现。（例如，在标准中没有指定指数的宽度。）单精度和双精度格式如下所示。



符号位s编码为0表示正，编码为1表示负。偏移指数e和小数f的最大有效位在它们的左侧。单精度和双精度表示的浮点值由下面所示的格式编码：

单精度格式

e	f	值
0	0	± 0
0	$\neq 0$	$\pm 2^{-126}(0.f)$
1 to 254	–	$\pm 2^{e-127}(1.f)$
255	0	$\pm\infty$
255	$\neq 0$	NaN

双精度格式

e	f	值
0	0	± 0
0	$\neq 0$	$\pm 2^{-1022}(0.f)$
1 to 2046	–	$\pm 2^{e-1023}(1.f)$
2047	0	$\pm\infty$
2047	$\neq 0$	NaN

例如，考虑在单精度格式下的π的编码。在二进制[Knu1]中，

$\pi \approx 11.0010\ 0100\ 0011\ 1111\ 0110\ 1010\ 1000\ 1000\ 1000\ 0101\ 1010\ 0011\ 0000\ 10\cdots$

这在上表第三行给出的“规格化”数的范围内。由于前导1不存放于规格数的编码之中，所以π中的最大有效位的1被扔掉。为了使二进制小数点处于正确位置，指数e－127应该是1，因此e=128。因此，单精度表示如下：

```
0 10000000 10010010000111111011011
```

或者，用十六进制表示为：

```
40490FDB
```

其中，我们把这个小数四舍五入到了最接近的可表示数。

满足$1 \leqslant e \leqslant 254$的数称为“规格化数”。它们具有“规格化”形式，其含义就是它们的最大有效位没有被明确地存储起来。$e=0$的非零数称为“非规格化数”。它们的最大有效位被明确地存储起来。这一设计方法有时称为“逐级下溢”。表15-1列出了浮点数各种范围内的一些极值。在这个表中，“最大整数”是指所有绝对值小于或等于它的整数都能精确表示，而下一个整数则被四舍五入的最大的整数。

表15-1 极值表

	十六进制	精确值	近似值
单精度			
最小非规格化数	0000 0001	2^{-149}	1.401×10^{-45}
最大非规格化数	007F FFFF	$2^{-126}(1-2^{-23})$	1.175×10^{-38}
最小规格化数	0080 0000	2^{-126}	1.175×10^{-38}
1.0	3F80 0000	1	1
最大整数	4B80 0000	2^{24}	1.677×10^{7}
最大规格化数	7F7F FFFF	$2^{128}(1-2^{-24})$	3.403×10^{38}
∞	7F80 0000	∞	∞
双精度			
最小非规格化数	0...0001	2^{-1074}	4.941×10^{-324}
最大非规格化数	000F...F	$2^{-1022}(1-2^{-52})$	2.225×10^{-308}
最小规格化数	0010...0	2^{-1022}	2.225×10^{-308}
1.0	3FF0...0	1	1
最大整数	4340...0	2^{53}	9.007×10^{15}
最大规格化数	7FEF...F	$2^{1024}(1-2^{-53})$	1.798×10^{308}
∞	7FF0...0	∞	∞

对于规格化数，在单精度格式的情况下，最后位置处的一个单位（ulp）有范围在$1/2^{24}$到$1/2^{23}$（约为5.96×10^{-8}到1.19×10^{-7}）的相对值，对于双精度格式，相对值的取值范围是$1/2^{53}$到$1/2^{52}$（约为1.11×10^{-16}到2.22×10^{-16}）。在四舍五入到最接近值的模式下，最大“相对误差”是这些值的一半。

对于单精度格式，能够精确表示的整数的范围是-2^{24}到$+2^{24}$（−16 777 216到+16 777 216），对于双精度格式，这一范围是-2^{53}到$+2^{53}$（−9 007 199 254 740 992到+9 007 199 254 740 992）。 262
当然，这些范围外的特定整数，例如更大的2的幂，也可被精确表示；所给出的范围是所有整数都可以精确表示的最大范围。

可能希望利用倒数把一个除以常量的除法转换成乘法。这种方法只能对倒数可以精确表示的那些数得到（IEEE）完全精确的结果。对于单精度格式，倒数可以精确表示的那些数是范围从2^{-127}到2^{127}的2的幂，对于双精度格式，它们是范围从2^{-1023}到2^{1023}的2的幂。数字2^{-127}和2^{-1023}的数是非规格化数，在不能有效实现非规格化数操作的计算机上，最好回避这些数。

15.2　利用整数操作进行浮点数比较

IEEE编码的特征之一就是，当把浮点数看成带符号整数时，非NaN值可以正确地排序。

为了使用整数操作编写浮点比较程序，我们需要忽视“不可排序”的结果。在IEEE 754中，当比较数之一或两个都是NaN时，发生不可排序的结果。下面的方法把NaN当作比无穷大还大的量来处理。

如果规定−0.0严格小于+0.0的话，那么比较会简单一些（这与IEEE 754不符）。假设这条 263
规定是可接受的，那么可以按如下方法完成比较，其中，$\overset{f}{<}$、$\overset{f}{\leqslant}$和$\overset{f}{=}$表示浮点比较，当公式不能正确处理±0.0时，我们用符号≈作为提示。

$$a \overset{f}{=} b \approx (a = b)$$

$$a \overset{f}{<} b \approx (a \geqslant 0 \;\&\; a < b) \mid (a < 0 \;\&\; a \overset{u}{>} b)$$

$$a \overset{f}{\leqslant} b \approx (a \geqslant 0 \;\&\; a \leqslant b) \mid (a < 0 \;\&\; a \overset{u}{\geqslant} b)$$

如果规定−0.0必须等于+0.0的话，那么似乎没有什么更好的方法实现比较，但是可以选择下面的公式，这些公式或多或少可以看成是由上面的公式得来的：

$$\begin{aligned}
a \overset{f}{=} b &\equiv (a = b) \mid (-a = a \;\&\; -b = b) \\
&\equiv (a = b) \mid ((a \mid b) = \mathbf{0x80000000}) \\
&\equiv (a = b) \mid (((a \mid b) \;\&\; \mathbf{0x7FFFFFFF}) = 0) \\
a \overset{f}{<} b &\equiv ((a \geqslant 0 \;\&\; a < b) \mid (a < 0 \;\&\; a \overset{u}{>} b)) \;\&\; ((a \mid b) \neq \mathbf{0x80000000}) \\
a \overset{f}{\leqslant} b &\equiv (a \geqslant 0 \;\&\; a \leqslant b) \mid (a < 0 \;\&\; a \overset{u}{\geqslant} b) \mid ((a \mid b) = \mathbf{0x80000000})
\end{aligned}$$

在某些应用中，下面的做法可能更加有效：首先用某种方法对这些数进行转换，然后使用单精度定点比较指令做浮点比较。例如，在给n个数排序时，对每个数只需做一次转换，而比较却至少要做$\lceil n\log_2 n \rceil$次。

表15-2给出了4种这样的转换。对于左栏中的那些转换，−0.0与+0.0的比较结果为相等，而对于右栏中的那些转换，−0.0与+0.0的比较结果为小于。在所有情况下，转换不改变比较的意义。变量n是带符号的，而t是无符号的，c可能是带符号的也可能是无符号的。

最后一行给出的是无分支代码，左侧的转换可以用4个基本RISC指令实现，右侧的转换可以用3个指令实现（对每个比较数都要执行这3个或4个指令）。

表15-2 使用整数比较时的浮点数的预处理

−0.0 = +0.0 (IEEE)	−0.0 < +0.0 (non-IEEE)

if (n >= 0) n = n+0x80000000;
else n = -n;
Use unsigned comparison.
 	
c = 0x7FFFFFFF;
if (n < 0) n = (n ^ c) + 1;
Use signed comparison.
 	
c = 0x80000000;
if (n < 0) n = c - n;
Use signed comparison.
 	
t = n >> 31;
n = (n ^ (t >> 1)) - t;
Use signed comparison.
	if (n >= 0) n = n+0x80000000;
	else n = ~n;
	Use unsigned comparison.
	
	c = 0x7FFFFFFF;
	if (n < 0) n = n ^ c;
	Use signed comparison.
	
	c = 0x7FFFFFFF;
	if (n < 0) n = c - n;
	Use signed comparison.
	
	t = (unsigned)(n>>30) >> 1;
	n = n ^ t;
	Use signed comparison.

15.3 前导数字分布

当1964年IBM推出System/360计算机时，数值分析员对单精度算法的精度损失感到非常恐惧。先前的IBM计算机系列，如704-709-7090家族，具有36位字。对于单精度浮点，格式由一
264 个9位的符号与指数字段及后面跟着的27位二进制小数组成。（在“规格化”数的范围）小数的最大有效位明确地包含在内，所以数的精度是27位。

S/360的字长是32位。对于单精度，IBM选择了8位符号与指数字段，后面是一个24位小数。精度从27位降到24位已经够糟了，但是还有更糟的。为了保持大的指数范围，S/360格式的7位指数的每个单元代表16的因子数。因此，小数是以16为底的，这种格式被称之为“十六进制”浮点。前导数字可以是1到15间的任意数（二进制为0001到1111）。带有前导数字1的数的精度仅为21位（因为有三个前导0），但是它们只占所有数的1/15（6.7%）。

不，比这更糟！一系列的分析和经验都表明，前导数字不是均匀分布的。在十六进制浮点中，前导位为1的数，即，精确度为21位的数的期待值是25%。

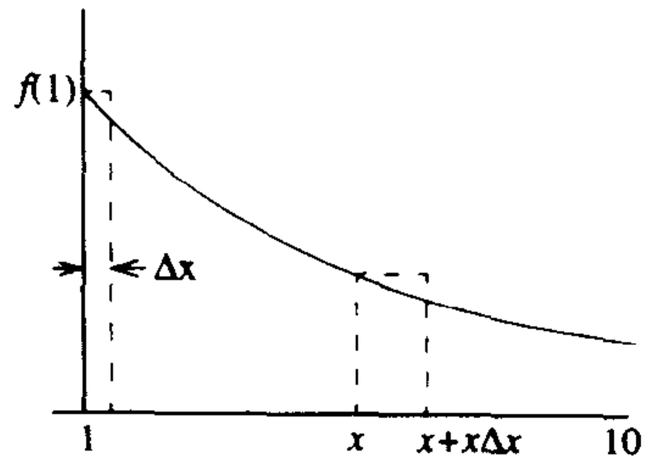
让我们考虑十进制中的前导数字的分布情况。假设有一个巨大的带单位的数集，单位可以是长度、体积、质量、速度等等，这些数是用“科学”表示法表示的（例如，6.022×10^{23}）。如果这样的数的大部分的前导数字有明确的分布函数，那么它必须与单位无关，即不管是英寸还是厘米，是磅还是千克等等。因此，如果用任意一个常量乘以集合中的所有数，前导数字的分布将不会改变。例如，考虑乘以2的乘法，我们得到的结论是带有前导数字1的数（从1.0到1.999…乘以10的若干次方）的数目必须等于带有前导数字2或3的数（从2.0到3.999…乘以10的若干次方）的数目，因为长度单位是一英寸还是半英寸，或者质量单位是千克还是半
265 千克等等都不影响分布。

对于$1 < x < 10$，设$f(x)$是带单位的数集的前导数字的概率密度函数。$f(x)$有如下性质：

$$\int_a^b f(x)\mathrm{d}x$$

它与前导数字在a和b之间的数成比例。参见下图，对x的小增量Δx，f必须满足下面的公式：

$$f(1) \cdot \Delta x = f(x) \cdot x\Delta x$$



因为$f(1) \cdot \Delta x$是范围在1到$1+\Delta x$内（忽略10的幂的乘数）的数的近似比例，而且$f(x) \cdot x\Delta x$是范围在x到$x + x\Delta x$之间的数的近似比例。因为后者的集合是前者的集合乘以x，它们的比例一定相同。因此，概率密度函数是简单的倒数关系：

$$f(x) = f(1)/x$$

因为，曲线在x=1到x=10区间内的面积一定等于1（所有数的前导位都在1.000…到9.999…之间），很容易证明：

$$f(1) = 1/\ln 10$$

对于$1 < a < b < 10$，前导数字在a到b之间的数的比例如下：

$$\int_a^b \frac{dx}{x\ln 10} = \frac{\ln x}{\ln 10}\Big|_a^b = \frac{\ln b/a}{\ln 10} = \log_{10}\frac{b}{a}$$

因此，在十进制中，前导数字是1的数的比例是$\log_{10}(2/1) \approx 0.30103$，前导数字是9的数的比例是$\log_{10}(10/9) \approx 0.0458$。 266

类似地，对于以16为底的情况，对于$1 < a < b < 16$，前导数字在a到b之间的数的比例是$\log_{16}(b/a)$。因此前导数字是1的数的比例是$\log_{16}(2/1) = 1/\log_2 16 = 0.25$。

15.4　各种各样的值的列表

表15-3给出了各种各样令人感情趣的值的IEEE表示。其中，不能精确表示的值被四舍五入到最接近的可表示值。

IEEE 754没有给出如何区分发出信号的NaN和静止的NaN。表15-3使用了PowerPC、AMD 29050、 Intel x86和I860及Fairchild Clipper所使用的表记方式。这一表记是：对于发出信号的NaN，小数的最大有效数位是0，对于静止的NaN，小数的最大有效数位是1。Compaq Alpha 、HP PA-RISC和 MIPS计算机使用同样的位加以区分，但是采用相反的表记方式（即静止=0，信号=1）。

表15-3　各种各样的值

十进制表示	单精度格式(十六进制)	双精度格式（十六进制）
$-\infty$	FF80 0000	FFF0 0000 0000 0000
-2.0	C000 0000	C000 0000 0000 0000
-1.0	BF80 0000	BFF0 0000 0000 0000

（续）

十进制表示	单精度格式（十六进制）	双精度格式（十六进制）
−0.5	BF00 0000	BFE0 0000 0000 0000
−0.0	8000 0000	8000 0000 0000 0000
+0.0	0000 0000	0000 0000 0000 0000
Smallest positive denorm	0000 0001	0000 0000 0000 0001
Largest denorm	007F FFFF	000F FFFF FFFF FFFF
Least positive normalized	0080 0000	0010 0000 0000 0000
$\pi/180$ (0.01745...)	3C8E FA35	3F91 DF46 A252 9D39
0.1	3DCC CCCD	3FB9 9999 9999 999A
$\log_{10} 2$ (0.3010...)	3E9A 209B	3FD3 4413 509F 79FF
$1/e$ (0.3678...)	3EBC 5AB2	3FD7 8B56 362C EF38
1/ln 10 (0.4342...)	3EDE 5BD9	3FDB CB7B 1526 E50E
0.5	3F00 0000	3FE0 0000 0000 0000
ln 2 (0.6931...)	3F31 7218	3FE6 2E42 FEFA 39EF
$1/\sqrt{2}$ (0.7071...)	3F35 04F3	3FE6 A09E 667F 3BCD
1/ln 3 (0.9102...)	3F69 0570	3FED 20AE 03BC C153
1.0	3F80 0000	3FF0 0000 0000 0000
ln 3 (1.0986...)	3F8C 9F54	3FF1 93EA 7AAD 030B
$\sqrt{2}$ (1.414...)	3FB5 04F3	3FF6 A09E 667F 3BCD
1/ln 2 (1.442...)	3FB8 AA3B	3FF7 1547 652B 82FE
$\sqrt{3}$ (1.732...)	3FDD B3D7	3FFB B67A E858 4CAA
2.0	4000 0000	4000 0000 0000 0000
ln 10 (2.302...)	4013 5D8E	4002 6BB1 BBB5 5516
e (2.718...)	402D F854	4005 BF0A 8B14 5769
3.0	4040 0000	4008 0000 0000 0000
π (3.141...)	4049 0FDB	4009 21FB 5444 2D18
$\sqrt{10}$ (3.162...)	404A 62C2	4009 4C58 3ADA 5B53
$\log_2 10$ (3.321...)	4054 9A78	400A 934F 0979 A371
4.0	4080 0000	4010 0000 0000 0000
5.0	40A0 0000	4014 0000 0000 0000
6.0	40C0 0000	4018 0000 0000 0000
2π (6.283...)	40C9 0FDB	4019 21FB 5444 2D18
7.0	40E0 0000	401C 0000 0000 0000
8.0	4100 0000	4020 0000 0000 0000
9.0	4110 0000	4022 0000 0000 0000
10.0	4120 0000	4024 0000 0000 0000
11.0	4130 0000	4026 0000 0000 0000
12.0	4140 0000	4028 0000 0000 0000

（续）

十进制表示	单精度格式（十六进制）	双精度格式（十六进制）
13.0	4150 0000	402A 0000 0000 0000
14.0	4160 0000	402C 0000 0000 0000
15.0	4170 0000	402E 0000 0000 0000
16.0	4180 0000	4030 0000 0000 0000
180/π (57.295...)	4265 2EE1	404C A5DC 1A63 C1F8
$2^{23}-1$	4AFF FFFE	415F FFFF C000 0000
2^{23}	4B00 0000	4160 0000 0000 0000
$2^{24}-1$	4B7F FFFF	416F FFFF E000 0000
2^{24}	4B80 0000	4170 0000 0000 0000
$2^{31}-1$	4F00 0000	41DF FFFF FFC0 0000
2^{31}	4F00 0000	41E0 0000 0000 0000
$2^{32}-1$	4F80 0000	41EF FFFF FFE0 0000
2^{32}	4F80 0000	41F0 0000 0000 0000
2^{52}	5980 0000	4330 0000 0000 0000
2^{63}	5F00 0000	43E0 0000 0000 0000
2^{64}	5F80 0000	43F0 0000 0000 0000
最大规格化数	7F7F FFFF	7FEF FFFF FFFF FFFF
∞	7F80 0000	7FF0 0000 0000 0000
“最小”发出信号的NaN	7F80 0001	7FF0 0000 0000 0001
“最大”发出信号的NaN	7FBF FFFF	7FF7 FFFF FFFF FFFF
“最小”静止NaN	7FC0 0000	7FF8 0000 0000 0000
“最大”静止NaN	7FFF FFFF	7FFF FFFF FFFF FFFF

267
~
269

第16章　素数公式

16.1　介绍

像很多年轻学生一样，我曾经对素数很着迷，试图找出一些关于素数的公式。我不能确定在一个公式中什么样的操作是有效的，也不能确定我要找的是什么样的函数，是求第n个素数或前n个素数的公式，还是能生成部分素数的公式，等等。不过，尽管这是一个含糊不清的问题，我仍然喜欢讨论这一问题的一些已知知识。我们将看到：存在素数公式；这些公式中没有令人满意的。

这一论题的大部分内容类似于程序设计技巧中对所用公式的处理，只是这里考虑的是实数范围的算术而不是“计算机算术”。但是，让我们回顾一下历史上有关这一课题的精彩部分。

1640年，Fermat猜想下面的公式

$$F_n = 2^{2^n} + 1$$

总能生成素数，这一形式的素数后来被称之为“Fermat数”。当n的取值范围是0到4时，F_n是素数的事实成立，但是Euler在1732年发现

$$F_5 = 2^{2^5} + 1 = 641 \times 6700417$$

（在32位计算机上实现除以常量的除法时看到过这些因子）。接着，F.Landry在1880年给出

$$F_6 = 2^{2^6} + 1 = 274177 \times 67280421310721$$

现在我们知道，对于许多较大的n，F_n是合数，例如所有7到16（包括这两个数）之间的n。对于$n>4$的数，至今还不知道有素数存在[H&W]。这是一个多么轻率的猜想[⊖]！ 271

顺便提一句，为什么Fermat使用双重指数呢？他知道，如果m有一个不等于1的奇数因子，那么2^m+1是合数。因为，对于所有$m=ab$且b是不等于1的奇数，有：

$$2^{ab} + 1 = (2^a + 1)(2^{a(b-1)} - 2^{a(b-2)} + 2^{a(b-3)} - \ldots + 1)$$

知道了这一事实，他一定想知道当m不含任何不等于1的奇数因子时，即$m=2^n$时，2^m+1如何呢。他尝试了n的几个值，发现$2^{2^n}+1$似乎应该是素数。

无疑，每个人都同意多项式有资格作为“公式”。1772年Leonhard Euler发现了一个相当令人惊讶的多项式。他发现，对0到39之间的每一个n，

$$f(n) = n^2 + n + 41$$

都是素数。可以对他的这一结果进行扩展。因为

⊖ 然而，这是已知Fermat猜想中惟一猜错的一个猜想[Wells]。

$$f(-n) = n^2 - n + 41 = f(n-1)$$

对于0到40之间的每一个n，$f(-n)$是素数；也就是说，对于-1到-40之间的每个n，$f(n)$是素数。因此，对于0到79之间的每个n，

$$f(n-40) = (n-40)^2 + (n-40) + 41 = n^2 - 79n + 1601$$

是素数。（然而，这一多项式不够美观，因为它是非单调的，而且还有重复出现；即，对于n=0，1，…，79，$n^2-79n+1601$=1601，1523，1447，…，43，41，41，43，…，1447，1523，1601。）

尽管有这样的成功公式，但是，现在已经知道，不存在对每一个n都生成素数的多项式$f(n)$（如$f(n)$=5这样的常量多项式除外）。事实上，任意非平凡“指数多项式”总有无穷多个合数值。更精确地说，如下定理成立[H&W]，

定理 如果$f(n) = P(n, 2^n, 3^n, \ldots, k^n)$是其参数的整系数的多项式，且当$n \to \infty$时，$f(n) \to \infty$，那么存在无穷多个使$f(n)$为合数的$n$。

因此，一个诸如$n^2 \cdot 2^n + 2n^3 + 2n + 5$的公式一定生成无穷多个合数。另一方面，这一定理对包含诸如项2^{2^n}、n^n和$n!$的多项式没有做任何说明。

利用地板函数和下面的魔术数可以求得第n个素数：

272

$$a = 0.203005000700011000013\ldots$$

a是这样的一个数：在十进制中，小数点后第一位是第一个素数，其后的两位是第二个素数，其后的三位是第三个素数，以此类推。对于任意的n，总有存放第n个素数p_n的空间，因为$p_n < 10^n$。我们对此不做证明，只是指出，（对于$n \geqslant 2$）已知n和$2n$之间一定存在素数，因此，在n到$10n$之间至少存在一个素数，因此有$p_n < 10^n$。求第n个素数的公式是：

$$p_n = \left\lfloor 10^{\frac{n^2+n}{2}} a \right\rfloor - 10^n \left\lfloor 10^{\frac{n^2-n}{2}} a \right\rfloor$$

其中，我们使用了关系$1+2+3+\cdots+n=(n^2+n)/2$。例如，有

$$\begin{aligned} p_3 &= \lfloor 10^6 a \rfloor - 10^3 \lfloor 10^3 a \rfloor \\ &= 203005 - 203000 \\ &= 5 \end{aligned}$$

因为需要知道素数的结果来定义a，所以这是一个没有太大价值的技巧。如果有方法定义一个与素数无关的a，那么这一公式就有意义多了。但是，没人知道这样的定义。

显然，这一技术可以用来得到很多序列的公式，但是它提出了问题。

16.2 Willans公式

C. P Willans给出了下面求第n个素数的公式[Will]：

$$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n \left(\sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)! + 1}{x} \right\rfloor \right)^{-1/n}} \right\rfloor$$

这一公式得之于Wilson定理，Wilson定理指出：p是素数或1当且仅当$(p-1)! = -1 \pmod p$。因此，如果x为素数或1，则

$$\frac{(x-1)!+1}{x}$$

是整数，而对所有合数x，它是一个小数。因此有

$$F(x) = \left\lfloor \cos^2 \pi \frac{(x-1)!+1}{x} \right\rfloor = \begin{cases} 1, & x \text{ 为素数或1} \\ 0, & x \text{ 为合数} \end{cases} \tag{16-1}$$

273

因此，如果$\pi(m)$表示[⊖]小于等于m的素数的数目，那么有：

$$\pi(m) = -1 + \sum_{x=1}^{m} F(x) \tag{16-2}$$

注意有$\pi(p_n)=n$，进一步有：

对于$m<p_n$，$\pi(m) < n$，且

对于$m \geqslant p_n$，$\pi(m) \geqslant n$

因此，满足$\pi(m)<n$的$m>0$的数目是p_n-1。也就是说，

$$p_n = 1 + \sum_{m=1}^{\infty} (\pi(m) < n) \tag{16-3}$$

这里，$\pi(m)<n$是一个（0/1值）“谓词表达式”。

因为我们有求$\pi(m)$的公式，等式(16-3)给出了一个作为n的函数的求第n素数的公式。但是，它有两个令人无法接受的性质：无穷求和及使用谓词表达式，这不是标准的数学用法。

已经证明，对于$n>1$，n到$2n$之间至少存在一个素数。因此，小于等于2^n的素数的数目至少是n，也就是说，$\pi(2^n) \geqslant n$。因此，对于$m > 2^n$，谓词$\pi(m)<n$为0，所以上面的求和的上限可以用2^n替换。

Willans有一个更聪明的替换谓词表达式的方法。设

$$\text{对于 } x = 0, 1, 2, \ldots \text{及 } y = 1, 2, \ldots, \quad \mathrm{LT}(x, y) = \left\lfloor \sqrt[y]{\frac{y}{1+x}} \right\rfloor$$

那么，如果$x<y$，则$1 < y/(1+x) < y$，所以$1 < \sqrt[y]{y/(1+x)} < \sqrt[y]{y} < 2$。进一步，如果$x \geqslant y$，则$0<y/(1+x)<1$，所以$0 < \sqrt[y]{y/(1+x)} <1$。使用地板函数，我们有

$$\mathrm{LT}(x, y) = \begin{cases} 1, \text{ 对于 } x < y \\ 0, \text{ 对于 } x \geqslant y \end{cases}$$

也就是说，$\mathrm{LT}(x,y)$是谓词$x<y$（x和y在给定的范围取值）。

274

经过替换，等式(16-3)可以写成：

$$\begin{aligned} p_n &= 1 + \sum_{m=1}^{2^n} \mathrm{LT}(\pi(m), n) \\ &= 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{\frac{n}{1+\pi(m)}} \right\rfloor \end{aligned}$$

⊖ 我们不得不把符号π用于两种不同的目的，$\pi(m)$是标准的表记方式，应该不会引起混淆。

进一步，用等式(16-2)将上式中的π(*m*)替换成*F*(*x*)的形式，再用等式(16-1)替换*F*(*x*)，就可得到本节开始给出的公式。

1. 第二个公式

Willans接着给出了另外一个公式：

$$p_n = \sum_{m=1}^{2^n} mF(m)\lfloor 2^{-|\pi(m)-n|} \rfloor$$

其中，*F*和π是第一个公式所用的函数。因此，如果*m*是素数或者是1，那么有*mF*(*m*)=*m*，否则*mF*(*m*)=0。公式的被加数中的第二个因子是谓词π(*m*)=*n*。被加数中，除了第*n*个素数外，其他项都是0。例如，

$$\begin{aligned} p_4 &= 1\times 1\times 0 + 2\times 1\times 0 + 3\times 1\times 0 + 4\times 0\times 0 + 5\times 1\times 0 + 6\times 0\times 0 + 7\times 1\times 1 \\ &\quad + 8\times 0\times 1 + 9\times 0\times 1 + 10\times 0\times 1 + 11\times 1\times 0 + \ldots + 16\times 0\times 0 \\ &= 7 \end{aligned}$$

2. 第三个公式

Willans继续给出了另一个求第*n*个素数的公式，这一公式不使用任何“非解析”[⊖]函数，例如地板函数和绝对值函数。他注意到，对*x*=2、3、…、有函数：

$$\frac{((x-1)!)^2}{x} = \begin{cases} \text{一个整数} + \dfrac{1}{x}, & \text{当}x\text{为素数时} \\ \text{一个整数}, & \text{当}x\text{为合数或1时} \end{cases}$$

275

由Wilson定理，这个函数的第一部分可以从下面的表达式及(*x*−1)!+1整除*x*得到：

$$\frac{((x-1)!)^2}{x} = \frac{((x-1)!+1)\cdot((x-1)!-1)}{x} + \frac{1}{x}$$

因此，对于*x*≥2，谓词“*x*是素数”可以从下面的式子得到：

$$H(x) = \frac{\sin^2\pi\dfrac{((x-1)!)^2}{x}}{\sin^2\dfrac{\pi}{x}}$$

由这一表达式，有

$$\text{对于}m = 2, 3, \ldots, \quad \pi(m) = \sum_{x=2}^{m} H(x)$$

用前面两个公式所用的方法不能将这个公式转换成求p_n的公式，因为它们使用了地板函数。取而代之的是，对于*x*,*y*≥1，Willans提出了下面关于谓词*x*<*y*的公式[⊜]：

$$\mathrm{LT}(x, y) = \sin\left(\frac{\pi}{2}\cdot 2^e\right), \quad \text{其中}$$

$$e = \prod_{i=0}^{y-1}(x-i)$$

⊖ 这是我的术语，不是Willans的

⊜ 我们对这一公式做了少许简化。

因此，如果$x<y$，则$e=x(x-1)...(0)(-1)...(x-(y-1))=0$，所以$LT(x,y)=\sin(\pi/2)=1$。如果$x>y$，则积不包含0，所以$e>1$，因此$LT(x,y)=\sin((\pi/2)\times(\text{一个偶数}))=0$。

最后，同Willans第一公式一样，有

$$p_n = 2 + \sum_{m=2}^{2^n} LT(\pi(m), n)$$

276

把这些全部写出来就得到了一个可怕的公式：

$$p_n = 2 + \sum_{m=2}^{2^n} \sin\left(\frac{\pi}{2}\times 2^{\prod_{i=0}^{n-1}\left(\sum_{x=2}^{m}\frac{\sin^2\pi\frac{((x-1)!)^2}{x}}{\sin^2\frac{\pi}{x}}-i\right)}\right)$$

3. 第四个公式

Willans接着给出了用p_n求p_{n+1}的公式：

$$p_{n+1} = 1 + p_n + \sum_{i=1}^{2p_n}\prod_{j=1}^{i} f(p_n + j)$$

其中，对于$x>2$，$f(x)$是谓词“x是合数”；也就是说，

$$f(x) = \left\lfloor \cos^2\pi\frac{((x-1)!)^2}{x} \right\rfloor$$

另外，这里也可以使用$f(x)=1-H(x)$来回避地板函数。

这个公式的一个例子如下，设$p_n=7$。那么有

$$\begin{aligned} p_{n+1} &= 1 + 7 + f(8) + f(8)f(9) + f(8)f(9)f(10) \\ &\quad + f(8)f(9)f(10)f(11) + \ldots + f(8)f(9)\ldots f(14) \\ &= 1 + 7 + 1 + 1\times 1 + 1\times 1\times 1 + 1\times 1\times 1\times 0 + \ldots + 1\times 1\times 1\times 0\times 1\times 0\times 1 \\ &= 11 \end{aligned}$$

16.3 Wormell公式

C.P Wormell[Wor]通过回避三角函数和地板函数改进了Willans的公式。理论上，可以使用只用整数算术的简单计算机程序来实现Wormell公式。他的推导不使用Wilson定理。对于$x>2$，Wormell以下面的函数开始他的推导：

$$B(x) = \prod_{a=2}^{x}\prod_{b=2}^{x}(x-ab)^2 = \begin{cases} \text{一个正整数，当}x\text{是素数时} \\ 0\text{，当}x\text{是合数时} \end{cases}$$

277

因此，小于等于m的素数数目可以由下面公式给出：

$$\pi(m) = \sum_{x=2}^{m}\frac{1+(-1)^{2B(x)}}{2}$$

这是因为被加数是谓词“x是素数”。

观察得到、对于$n \geqslant 1$、$a \geqslant 0$、有:

$$\prod_{r=1}^{n}(1-r+a)^2 \begin{cases} 0, & \text{当} a<n \text{时} \\ \text{一个正整数}, & \text{当} a \geqslant n \text{时} \end{cases}$$

重复使用上面的技巧、谓词$a<n$是:

$$(a<n) = \frac{1-(-1)^{2\prod_{r=1}^{n}(1-r+a)^2}}{2}$$

因为

$$p_n = 2 + \sum_{m=2}^{2^n}(\pi(m)<n)$$

通过把被加数中的常量分解出来、可得

$$p_n = \frac{3}{2} + 2^{n-1} - \frac{1}{2} \times \sum_{m=2}^{2^n}(-1)^{2\prod_{r=1}^{n}\left(1-r+\frac{(m-1)!}{2}+\frac{1}{2}\cdot\sum_{t=2}^{m}(-1)^{2\prod_{a=2}^{t}\prod_{b=2}^{t}(t-ab)^2}\right)^2}$$

如上所述、Wormell公式没有使用三角函数和地板函数。然而、正如他所指出的那样，如果使用$(-1)^n = \cos \pi n$来展开-1的幂、那么三角函数和地板函数就会再次出现。

16.4 求其他比较麻烦的函数的公式

让我们再进一步考察Willans和Wormell所做的工作。我们用下面的规则来定义可以用“公式”表示的函数、这些函数称为“公式函数”。这里，对于任意的$n \geqslant 1$，$\bar{x}$是$x_1, x_2, \ldots, x_n$的缩写。值域是整数$\cdots -2、-1、0、1、2、\cdots$

278

1) 常量$\cdots -1、0、1、\cdots$是公式函数。

2) 对于$1 \leqslant i \leqslant n$、投影函数$f(\bar{x})=x_i$是公式函数。

3) 如果x和y是公式函数、那么表达式$x+y$、$x-y$和xy是公式函数。

4) 公式函数族对于合成（替换）封闭。也就是说、如果f和g_i是公式函数（$i=1, \cdots, m$），那么$f(g_1(\bar{x}), g_2(\bar{x}), \ldots, g_m(\bar{x}))$也是公式函数。

5) 若a、b和f是公式函数、且$a(\bar{x}) \leqslant b(\bar{x})$，那么有界的和与积公式、写作:

$$\sum_{i=a(\bar{x})}^{b(\bar{x})}{}' f(i,\bar{x}) \quad \text{及} \quad \prod_{i=a(\bar{x})}^{b(\bar{x})}{}' f(i,\bar{x}),$$

它们也是公式函数。

和与积有界的条件是为了保持公式的计算性质；也就是说，通过给参数赋值且执行有限步的运算来计算公式。本章后半部分将给出要求和与积的界区间非空的理由。

当使用合成来构造新的公式函数时，按照惯例，在必要的时候可以使用括号。

注意、在上面的列表中没有包含除法；除法太复杂，不能轻易当作“公式函数”。即便如此、上面的定义不是极小的、例如、如果x是公式函数，那么既可以从第3条也可以从第5条得到公式函数$x+x$。找到公式函数的极小定义是有趣的，但在这里我们不加详述。

“公式函数”的定义与[Cut]给出的“初等函数”的定义很接近。然而，[Cut]中所使用的值域是非负整数（这在递归函数理论中是常见的）。同样，[Cut]要求累加和与累积的上下界分别为0和$x-1$（这里x是一个变量），而且允许界区间为空（对于这种情况，和被定义为0，积被定义为1）。

下面，我们要说明公式函数族所包含的范围相当大，包括数学中常见的大部分函数。但是，它不包含所有容易定义并且有初等函数性质的函数。

与递归函数理论的类似展开相比，我们的展开有一些障碍，原因是这里的变量可以取负值。然而，负值的可能性可以通过对某个一次幂表达式取平方来调整。累加和与累积的界区间非空的要求也会给我们带来一些麻烦。 279

这里，“谓词”简单指0/1值函数，而在递归函数理论中，谓词是真/假值函数，每一个谓词有一个相关的0/1值“特征函数”。我们把真与1相关联，假与0相关联，就像在程序设计语言和计算机上所做的那样（在那里，与和或指令就是这么做的）；在逻辑函数和递归函数理论中，这种关联通常是相反的。

下面是公式函数：

1) $a^2 = aa$，$a^3 = aaa$，以此类推。

2) 谓词$a=b$是 $(a = b) = \prod_{j=0}^{(a-b)^2}{}'(1-j)$。

3) $(a \neq b) = 1-(a=b)$。

4) 谓词$a \geq b$是 $(a \geq b) = \sum_{i=0}^{(a-b)^2}{}'((a-b) = i) = \sum_{i=0}^{(a-b)^2}{}' \prod_{j=0}^{((a-b)-i)^2}{}'(1-j)$。

现在能够解释为什么我们不使用空区间累加取值0和空区间累积取值1之约定的原因了。假如这样做了的话，我们将会得出如下不真实的结果：

$$(a = b) = \sum_{i=0}^{-(a-b)^2} 1 \qquad \text{及} \qquad (a \geq b) = \prod_{i=a}^{b-1} 0$$

比较谓词是下面公式的关键，我们不希望有任何人为的比较谓词公式。

5) $(a>b) = (a \geq b+1)$。

6) $(a < b) = (b \geq a)$。

7) $(a<b) = (b>a)$。

8) $|a| = (2(a \geq 0)-1)a$。

9) $\max(a, b) = (a \geq b)(a-b)+b$。

10) $\min(a, b)= (a \geq b)(b-a)+a$。

现在我们处理累加和与累积，使得当界区间为空时它们给出常规且有用的结果。 280

11) $\sum_{i=a(\bar{x})}^{b(\bar{x})} f(i, \bar{x}) = (b(\bar{x}) \geq a(\bar{x})) \sum_{i=a(\bar{x})}^{\max(a(\bar{x}), b(\bar{x}))}{}' f(i, \bar{x})$。

12) $\prod_{i=a(\bar{x})}^{b(\bar{x})} f(i, \bar{x}) = 1 + (b(\bar{x}) \geq a(\bar{x}))(-1 + \prod_{i=a(\bar{x})}^{\max(a(\bar{x}), b(\bar{x}))}{}' f(i, \bar{x}))$。

从现在开始，我们将使用不带撇（'）的Σ与Π。这样，下面定义的所有函数都是全函数（对于参数的所有值都有定义）。

13) $n! = \prod_{i=1}^{n} i$。

对于$n<0$，这个公式给出$n!=1$。

下面的P和Q表示谓词。

14) $\neg P(\bar{x}) = 1 - P(\bar{x})$。

15) $P(\bar{x}) \,\&\, Q(\bar{x}) = P(\bar{x})Q(\bar{x})$。

16) $P(\bar{x}) \mid Q(\bar{x}) = 1 - (1 - P(\bar{x}))(1 - Q(\bar{x}))$。

17) $P(\bar{x}) \oplus Q(\bar{x}) = (P(\bar{x}) - Q(\bar{x}))^2$。

18) if $P(\bar{x})$ then $f(\bar{y})$ else $g(\bar{z}) = P(\bar{x})f(\bar{y}) + (1 - P(\bar{x}))g(\bar{z})$。

19) a^n = if $n \geqslant 0$ then $\prod_{i=1}^{n} a$ else 0。

对于$n<0$，这个公式定义结果为0，而0^0为1。这有些武断，可能对某些情况是不正确的。

20) $(m \leqslant \forall x \leqslant n)P(x, \bar{y}) = \prod_{x=m}^{n} P(x, \bar{y})$。

21) $(m \leqslant \exists x \leqslant n)P(x, \bar{y}) = 1 - \prod_{x=m}^{n} (1 - P(x, \bar{y}))$。

∀是空真（vacuously true），∃是空假（vacuously false）。

22) $(m \leqslant \min x \leqslant n)P(x, \bar{y}) = m + \sum_{i=m}^{n} \prod_{j=m}^{i} (1 - P(j, \bar{y}))$。

当界区间为空时，这一表达式的值为m；否则，当谓词$P(x, \bar{y})$在整个（非空）界区间都为假时，表达式的值为$n+1$；否则，表达式的值为在m到n之间使得谓词$P(x, \bar{y})$为真的最小x。这一操作被称为“有界极小化”，是生成新公式函数的强有力工具。如下一个公式所示，这一操作是一类函数求逆操作。使用积的和实现极小化的方法是由Goodstein[Good]给出的。

281

23) $\lfloor \sqrt{n} \rfloor = (0 \leqslant \min k \leqslant |n|)\ ((k+1)^2 > n)$。

这是“整数平方根”函数，对于$n<0$，我们定义它的值为0。

24) $d|n = (-|n| \leqslant \exists q \leqslant |n|)\ (n=qd)$。

这是“d整除n”谓词，根据公式，0整除0，但对于$n \neq 0$，0不整除n。

25) $n \div d$ = if $n \geqslant 0$ then $(-n \leqslant \min q \leqslant n)(0 \leqslant \exists r \leqslant |d|-1)(n=qd+r)$

else $(n \leqslant \min q \leqslant -n)(-|d|+1 \leqslant \exists r \leqslant 0)(n=qd+r)$。

这是常见的整数除法的截取形式。对于$d=0$，它给出武断的结果$|n|+1$。

26) $\mathrm{rem}(n, d) = n-(n \div d)d$。

这是常见的余数函数。如果$\mathrm{rem}(n, d)$是非零的，那么它与n的符号相同，如果$d=0$，余数是n。

27) $\mathrm{isprime}(n) = n \geqslant 2 \,\&\, \neg\, (2 \leqslant \exists d \leqslant |n|-1)(d|n)$。

28) $\pi(n) = \sum_{i=1}^{n} \mathrm{isprime}(i)$。（小于等于$n$的素数的数目。）

29) $p_n = (1 \leqslant \min k \leqslant 2^n)(\pi(k)=n)$。

30) exponent$(p, n) = (0 \leqslant \min x \leqslant |n|) \neg (p^{x+1} | n)$。

对于$n \geqslant 1$，这一表达式给出n中素数因子p的指数。

31) 对于$n \geqslant 0$，

$$2^n = \prod_{i=1}^{n} 2, \qquad 2^{2^n} = \prod_{i=1}^{2^n} 2, \qquad 2^{2^{2^n}} = \prod_{i=1}^{2^{2^n}} 2, \text{ 以此类推。}$$

32) $\sqrt{2}$的十进制展开中小数点后的第n位数字：$\operatorname{rem}(\lfloor \sqrt{2 \times 10^{2n}} \rfloor, 10)$。

因此，公式函数族所包含的范围相当大。尽管下面的定理（至少）指出了它的局限：

定理 如果f是公式函数，那么存在一个常数k，使得：

$$f(\bar{x}) \leqslant 2^{2^{2^{\max(|x_1|, \ldots, |x_n|)}}} \Big\} k$$

其中，公式中有k个2。 282

可以通过每次运用本节开头讲的规则1~5都保持定理成立来证明这一定理。例如，如果$f(\bar{x})=c$（规则1），那么对某个h，有：

$$f(\bar{x}) \leqslant 2^{2^{2}} \Big\} h$$

其中，公式中有h个2。因此，由于$\max(|x_1|, |x_2|, \ldots, |x_n|) \geqslant 0$，有：

$$f(\bar{x}) \leqslant 2^{2^{2^{\max(|x_1|, \ldots, |x_n|)}}} \Big\} h + 2$$

对于$f(\bar{x})=x_i$（规则2），有$f(\bar{x}) \leqslant \max(|x_1|, |x_2|, \ldots, |x_n|)$，所以定理对$k=0$成立。

对于规则3，设

$$f(\bar{x}) \leqslant 2^{2^{2^{\max(|x_1|, \ldots, |x_n|)}}} \Big\} k_1 \qquad \text{及} \qquad g(\bar{x}) \leqslant 2^{2^{\cdot^{\cdot^{2^{\max(|x_1|, \ldots, |x_n|)}}}}} \Big\} k_2$$

那么，显然有：

$$f(\bar{x}) \pm g(\bar{x}) \leqslant 2 \times 2^{2^{2^{\max(|x_1|, \ldots, |x_n|)}}} \Big\} \max(k_1, k_2)$$

$$\leqslant 2^{2^{2^{\max(|x_1|, \ldots, |x_n|)}}} \Big\} \max(k_1, k_2) + 1$$

类似地，可以证明，对于$f(x, y)=xy$，定理成立。

规则4和规则5也能保持定理成立的证明稍稍有些繁琐，但是不难，在这里省略。

根据这一定理可证，下面的函数

$$f(x) = 2^{2^{\cdot^{\cdot^{2^x}}}} \Big\} x \tag{16-4}$$

不是公式函数，因为对于任意给定的k个2，当x足够大时，等式(16-4)的值大于定理中同一表达式的值。

对递归函数理论有兴趣的人，我们要指出等式(16-4)是原始递归函数。进一步，由原始递

归函数的定义，很容易直接证明所有公式函数都是原始递归函数。因此，公式函数族是原始递归函数的一个真子集。有兴趣的读者可以参考[Cut]。

总之，本节证明了不仅存在求第n个素数的初等函数公式，而且还有很多其他在数学中常
283 见的函数的公式。另外，我们的“公式函数”不基于三角函数、地板函数、绝对值函数、-1的幂，也不基于除法。惟一的伎俩就是使用了这样的事实：多个数的积是0当且仅当它们中有一个是0，它被用于谓词$a=b$的公式中。

然而，千真万确的是：一旦了解了这些公式，它们就不再有吸引力了。寻找素数的“有趣”公式将会持续下去。例如，[Rib]引用了W.H.Mills（1947年）的一个令人惊讶的定理，那就是，存在一个θ使得下面的表达式

$$\lfloor \theta^{3^n} \rfloor$$

对于所有大于等于1的n都取素数值。事实上，有无穷多个这样的值θ（例如，1.3063778838+和1.4537508625483+）。另外，不一定要指定“3”，用任何一个大于等于3的整数替换3（对于不同的θ），定理也成立。更好的是，如果在n^2到$(n+1)^2$之间总存在素数的话，那么用2替代3定理仍然成立。在n^2到$(n+1)^2$之间总存在素数这一断言几乎是显然的，但是对此还没有人给出证明。而且……好了，有兴趣的读者可以参考[Rib]和[Dud]，寻找这一类型更令
284 人着迷的公式。

附录A　四位计算机的算术表

在附录A所列出的表中，下划线表示带符号溢出。

表A-1　加法表

		0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
	0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
	2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
	3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
	4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
	5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
	6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
	7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
-8	8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
-7	9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
-6	A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
-5	B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
-4	C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
-3	D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
-2	E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
-1	F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

减法表（表A-2）假设$a-b$的进位位被设置为$a+\bar{b}+1$的进位位，所以进位等价于“非借位”。

表A-2　减法表（行-列）

		0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
	0	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
	1	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2
	2	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3
	3	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4
	4	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5
	5	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6
	6	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7
	7	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8
-8	8	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9
-7	9	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A
-6	A	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B
-5	B	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C
-4	C	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D
-3	D	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E
-2	E	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F
-1	F	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10

对于乘法表（表A-3和表A-4），溢出意味着结果不能表示成4位量。对于带符号乘法（表A-3），在这一意义下，这等价于8位结果的前五位不都是1或都是0。

表A-3 带符号乘法表

	0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	F8	F9	FA	FB	FC	FD	FE	FF
2	0	2	4	6	8	A	C	E	F0	F2	F4	F6	F8	FA	FC	FE
3	0	3	6	9	C	F	12	15	E8	EB	EE	F1	F4	F7	FA	FD
4	0	4	8	C	10	14	18	1C	E0	E4	E8	EC	F0	F4	F8	FC
5	0	5	A	F	14	19	1E	23	D8	DD	E2	E7	EC	F1	F6	FB
6	0	6	C	12	18	1E	24	2A	D0	D6	DC	E2	E8	EE	F4	FA
7	0	7	E	15	1C	23	2A	31	C8	CF	D6	DD	E4	EB	F2	F9
-8 8	0	F8	F0	E8	E0	D8	D0	C8	40	38	30	28	20	18	10	8
-7 9	0	F9	F2	EB	E4	DD	D6	CF	38	31	2A	23	1C	15	E	7
-6 A	0	FA	F4	EE	E8	E2	DC	D6	30	2A	24	1E	18	12	C	6
-5 B	0	FB	F6	F1	EC	E7	E2	DD	28	23	1E	19	14	F	A	5
-4 C	0	FC	F8	F4	F0	EC	E8	E4	20	1C	18	14	10	C	8	4
-3 D	0	FD	FA	F7	F4	F1	EE	EB	18	15	12	F	C	9	6	3
-2 E	0	FE	FC	FA	F8	F6	F4	F2	10	E	C	A	8	6	4	2
-1 F	0	FF	FE	FD	FC	FB	FA	F9	8	7	6	5	4	3	2	1

表A-4 无符号乘法表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

286

表A-5和表A-6是通常的截取除法的运算表。对于最大负数除以-1，表A-5给出了结果8和溢出，但是，大多数计算机对这一情况的结果未做定义，或者禁止这一操作。

表A-7和表A-8给出对应于截取除法的余数表。对于最大负数除以-1，表A-7给出了结果0，但是，大多数计算机对这一情况的结果未做定义，或者禁止这一操作。

表A-5 带符号短除法表（行÷列）

		0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	F
	2	-	2	1	0	0	0	0	0	0	0	0	0	0	0	F	E
	3	-	3	1	1	0	0	0	0	0	0	0	0	0	F	F	D
	4	-	4	2	1	1	0	0	0	0	0	0	0	F	F	E	C
	5	-	5	2	1	1	1	0	0	0	0	0	F	F	F	E	B
	6	-	6	3	2	1	1	1	0	0	0	F	F	F	E	D	A
	7	-	7	3	2	1	1	1	1	0	F	F	F	F	E	D	9
-8	8	-	8	C	E	E	F	F	F	1	1	1	1	2	2	4	8
-7	9	-	9	D	E	F	F	F	F	0	1	1	1	1	2	3	7
-6	A	-	A	D	E	F	F	F	0	0	0	1	1	1	2	3	6
-5	B	-	B	E	F	F	F	0	0	0	0	0	1	1	1	2	5
-4	C	-	C	E	F	F	0	0	0	0	0	0	0	1	1	2	4
-3	D	-	D	F	F	0	0	0	0	0	0	0	0	0	1	1	3
-2	E	-	E	F	0	0	0	0	0	0	0	0	0	0	0	1	2
-1	F	-	F	0	0	0	0	0	0	0	0	0	0	0	0	0	1

表A-6 无符号短除法表（行÷列）

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	-	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	-	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	-	4	2	1	1	0	0	0	0	0	0	0	0	0	0	0
5	-	5	2	1	1	1	0	0	0	0	0	0	0	0	0	0
6	-	6	3	2	1	1	1	0	0	0	0	0	0	0	0	0
7	-	7	3	2	1	1	1	1	0	0	0	0	0	0	0	0
8	-	8	4	2	2	1	1	1	1	0	0	0	0	0	0	0
9	-	9	4	3	2	1	1	1	1	1	0	0	0	0	0	0
A	-	A	5	3	2	2	1	1	1	1	1	0	0	0	0	0
B	-	B	5	3	2	2	1	1	1	1	1	1	0	0	0	0
C	-	C	6	4	3	2	2	1	1	1	1	1	1	0	0	0
D	-	D	6	4	3	2	2	1	1	1	1	1	1	1	0	0
E	-	E	7	4	3	2	2	2	1	1	1	1	1	1	1	0
F	-	F	7	5	3	3	2	2	1	1	1	1	1	1	1	1

表A-7 带符号短除法的余数表（行÷列）

	0	1	2	3	4	5	6	7	-8 8	-7 9	-6 A	-5 B	-4 C	-3 D	-2 E	-1 F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
2	-	0	0	2	2	2	2	2	2	2	2	2	2	2	0	0
3	-	0	1	0	3	3	3	3	3	3	3	3	3	0	1	0
4	-	0	0	1	0	4	4	4	4	4	4	4	0	1	0	0
5	-	0	1	2	1	0	5	5	5	5	5	0	1	2	1	0
6	-	0	0	0	2	1	0	6	6	6	0	1	2	0	0	0
7	-	0	1	1	3	2	1	0	7	0	1	2	3	1	1	0
-8 8	-	0	0	E	0	D	E	F	0	F	E	D	0	E	0	0
-7 9	-	0	F	F	D	E	F	0	9	0	F	E	D	F	F	0
-6 A	-	0	0	0	E	F	0	A	A	A	0	F	E	0	0	0
-5 B	-	0	F	E	F	0	B	B	B	B	B	0	F	E	F	0
-4 C	-	0	0	F	0	C	C	C	C	C	C	C	0	F	0	0
-3 D	-	0	F	0	D	D	D	D	D	D	D	D	D	0	F	0
-2 E	-	0	0	E	E	E	E	E	E	E	E	E	E	E	0	0
-1 F	-	0	F	F	F	F	F	F	F	F	F	F	F	F	F	0

表A-8 无符号短除法的余数表（行÷列）

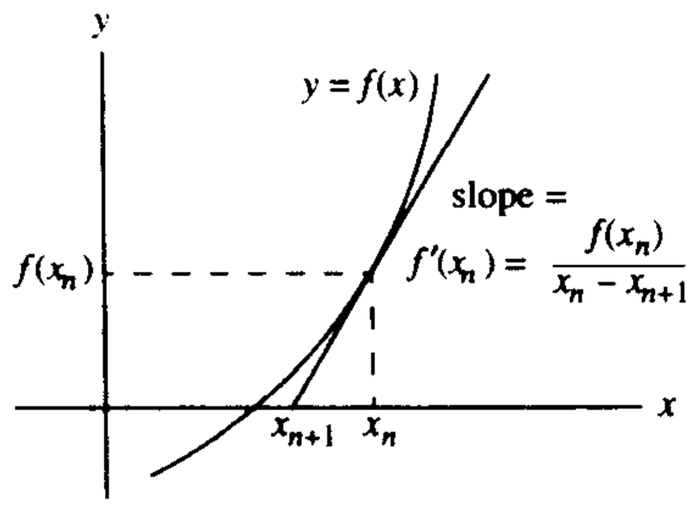
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	-	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
3	-	0	1	0	3	3	3	3	3	3	3	3	3	3	3	3
4	-	0	0	1	0	4	4	4	4	4	4	4	4	4	4	4
5	-	0	1	2	1	0	5	5	5	5	5	5	5	5	5	5
6	-	0	0	0	2	1	0	6	6	6	6	6	6	6	6	6
7	-	0	1	1	3	2	1	0	7	7	7	7	7	7	7	7
8	-	0	0	2	0	3	2	1	0	8	8	8	8	8	8	8
9	-	0	1	0	1	4	3	2	1	0	9	9	9	9	9	9
A	-	0	0	1	2	0	4	3	2	1	0	A	A	A	A	A
B	-	0	1	2	3	1	5	4	3	2	1	0	B	B	B	B
C	-	0	0	0	0	2	0	5	4	3	2	1	0	C	C	C
D	-	0	1	1	1	3	1	6	5	4	3	2	1	0	D	D
E	-	0	0	2	2	4	2	0	6	5	4	3	2	1	0	E
F	-	0	1	0	3	0	3	1	7	6	5	4	3	2	1	0

附录B　牛顿方法

这里，我们简单回顾一下牛顿方法。给定实变量x的可导函数f，我们求满足$f(x)=0$的x。给定f的根的当前估计值x_n，在适当的条件下，牛顿方法根据下面的公式给出更好的估计值x_{n+1}：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

这里，$f'(x_n)$是f在$x=x_n$处的导数。这一公式的导数可以从（求解x_{n+1}的）下图中读取。



这一方法适用于简单、具有良好性质的函数，例如多项式函数，只要第一个估计值足够接近。一旦估计值充分逼近，这个方法就二次收敛。也就是说，如果r是根的精确值，x_n是充分逼近的估计值，那么有

$$|x_{n+1} - r| < (x_n - r)^2$$

因此，每次迭代都使精确数字的数目成倍增加（例如，如果$|x_n - r| < 0.001$，那么$|x_{n+1} - r| < 0.0000001$）。

如果第一个估计值相差较远，那么迭代也许很慢地收敛，也许发散到无穷远处，也许收敛到一个不是最接近于第一个估计值的根，也许在某些特定的值之间无限循环。

上面的讨论由于某些措辞（例如“适当的条件”、“好的性质”和“充分逼近”）而显得含混不清。对于更精确的讨论，可以参考任何大学一年级的微积分课本。

尽管这一方法有上述的约束，但是，有时在整数范围内这一方法很有用。为了考察这一方法是否可以应用于某一特殊的函数上，读者必须进行尝试，正如11.1节中所做的那样。

表B-1给出了由牛顿方法得到的几个用来计算特定数的迭代公式。表中第一列给出要计算的数；第二列给出以第一列的数为根的函数；第三列是对应于第二列的函数的牛顿方法的公式右部。

顺便提一下，寻找好用的函数并不总是一件容易的事情。当然，有很多以期望值为根的函数，它们当中只有几个可以导出有用的迭代公式。通常，可用的函数是期望的计算的某种形式的逆运算。例如，要找$\sqrt{a}$则需要使用函数$f(x)=x^2-a$，求$\log_2 a$则需要使用函数$f(x)=2^x-a$，等等[⊖]。

⊖ 巴比伦人在大约4 000年就已经知道了求平方根函数的牛顿方法。

表B-1 计算特定数的牛顿方法

被计算的量	函数	迭代公式
$\sqrt{a}$	$x^2 - a$	$\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$
$\sqrt[3]{a}$	$x^3 - a$	$\frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right)$
$\frac{1}{\sqrt{a}}$	$x^{-2} - a$	$\frac{x_n}{2}(3 - ax_n^2)$
$\frac{1}{a}$	$x^{-1} - a$	$x_n(2 - ax_n)$
$\log_2 a$	$2^x - a$	$x_n + \frac{1}{\ln 2}\left(\frac{a}{2^{x_n}} - 1\right)$

$\log_2 a$的迭代公式收敛（到$\log_2 a$），即使对乘数1/ln2做某种程度的改变（例如，把1/ln2改成1或改成2）。然而，它的收敛会变得更慢。在某些应用中可能使用3/2或23/16更好（$1/\ln 2 \approx 1.4427$）。

290

参考文献

[AES] *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, FIPS PUB 197 (November 2001). Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[Alv] Alverson, Robert. "Integer Division Using Reciprocals." In *Proceedings IEEE 10th Symposium on Computer Arithmetic*, June 26–28, 1991, Grenoble, France, 186–190.

[Aus1] Found in a REXX interpreter subroutine written by Marc A. Auslander.

[Aus2] Auslander, Marc A. Private communication.

[Bern] Bernstein, Robert. "Multiplication by Integer Constants." *Software—Practice and Experience 16*, 7 (July 1986), 641–652.

[BGN] Burks, Arthur W., Goldstine, Herman H., and von Neumann, John. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Second Edition" (1947). In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[CJS] Stephenson, Christopher J. Private communication.

[Cohen] These rules were pointed out by Norman H. Cohen.

[Cut] Cutland, Nigel J. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[CWG] Hoxey, Karim, Hay, and Warren (Editors). *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.

[DES] *Data Encryption Standard (DES)*, National Institute of Standards and Technology, FIPS PUB 46-2 (December 1993). Available at http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[Dewd] Dewdney, A. K. *The Turing Omnibus*. Computer Science Press, 1989.

[Dud] Dudley, Underwood. "History of a Formula for Primes." *American Mathematics Monthly 76* (1969), 23–28.

[EL] Ercegovac, Miloš D. and Lang, Tomás. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[Gard] Gardner, Martin. "Mathematical Games" column in *Scientific American 227*, 2 (August 1972), 106–109.

[GGS] Gregoire, Dennis G., Groves, Randall D., and Schmookler, Martin S. *Single Cycle Merge/Logic Unit*, US Patent No. 4,903,228, February 20, 1990.

[GK] Granlund, Torbjörn and Kenner, Richard. "Eliminating Branches Using a Superoptimizer and the GNU C Compiler." In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), July 1992, 341–352.

[GKP] Graham, Ronald L., Knuth, Donald E., and Patashnik, Oren. *Concrete Mathematics: A Foundation for Computer Science, Second Edition*. Addison-Wesley, 1994.

[GLS1] Steele, Guy L., Jr. Private communication.

[GLS2] Steele, Guy L., Jr. "Arithmetic Shifting Considered Harmful." AI Memo 378, MIT Artificial Intelligence Laboratory (September 1976); also in *SIGPLAN Notices 12*, 11 (November 1977), 61–69.

[GM] Granlund, Torbjörn and Montgomery, Peter L. "Division by Invariant Integers Using Multiplication." In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (PLDI), August 1994, 61–72.

[Gold] The second expression is due to Richard Goldberg.

[Good] Goodstein, Prof. R. L. "Formulae for Primes." *The Mathematical Gazette 51* (1967), 35–36.

[GSO] Found by the GNU Superoptimizer.

[HAK] Beeler, M., Gosper, R. W., and Schroeppel, R. *HAKMEM*, MIT Artificial Intelligence Laboratory AIM 239, February 1972.

[Hay1] Hay, R. W. Private communication.

[Hay2] The first expression was found in a compiler subroutine written by R. W. Hay.

[Hil] Hilbert, David. "Ueber die stetige Abbildung einer Linie auf ein Flächenstück." *Mathematischen Annalen 38* (1891), 459–460.

[Hop] Hopkins, Martin E. Private communication.

[HS] Hillis, W. Daniel and Steele, Guy L., Jr. "Data Parallel Algorithms." *Comm. ACM 29*, 12 (December 1986) 1170–1183.

[H&P] Hennessy, John L. and Patterson, David A. *Computer Architecture: A*

Quantitative Approach. Morgan Kaufmann, 1990.

[H&S] Harbison, Samuel P. and Steele, Guy L., Jr. *C: A Reference Manual*, Fourth Edition. Prentice-Hall, 1995.

[H&W] Hardy, G. H. and Wright, E. M. *An Introduction to the Theory of Numbers*, Fourth Edition. Oxford University Press, 1960.

[IBM] From an IBM programming course, 1961.

[Irvine] Irvine, M. M. "Early Digital Computers at Bell Telephone Laboratories." *IEEE Annals of the History of Computing 23*, 3 (July-September 2001), 22–42.

[JVN] von Neumann, John. "First Draft of a Report on the EDVAC." In *Papers of John von Neumann on Computing and Computing Theory*, Volume 12 in the Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1987.

[Ken] Found in a GNU C compiler for the RS/6000 that was ported by Richard Kenner. He attributes this to a 1992 PLDI conference paper by him and Torbjörn Granlund.

[Knu1] Knuth, Donald E. *The Art of Computer Programming, Volume 1, Third Edition: Fundamental Algorithms*. Addison-Wesley, 1997.

[Knu2] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Third Edition: Seminumerical Algorithms*. Addison-Wesley, 1998.

[Knu3] The idea of using a negative integer as the base of a number system for arithmetic has been independently discovered by many people. The earliest reference given by Knuth is to Vittorio Grünwald in 1885. Knuth himself submitted a paper on the subject in 1955 to a "science talent search" for high-school seniors. For other early references, see Knuth, Volume 2.

[KRS] Kruskal, Clyde P., Rudolph, Larry, and Snir, Marc. "The Power of Parallel Prefix." *IEEE Transactions on Computers C-34*, 10 (October 1985), 965–968.

[Lamp] Lamport, Leslie. "Multiple Byte Processing with Full-Word Instructions." *Communications of the ACM 18*, 8 (August 1975), 471–475.

[LSY] Lee, Ruby B., Shi, Zhijie, and Yang, Xiao. "Efficient Permutation Instructions for Fast Software Cryptography." *IEEE Micro 21*, 6 (November/December 2001), 56–69.

[L&S] Lam, Warren M. and Shapiro, Jerome M. "A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve." In *Proceedings ICIP 94*, 1 (1994), 638–641.

[MD] Denneau, Monty. Private communication.

[MIPS] Kane, Gerry and Heinrich, Joe. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[MM] Morton, Mike. "Quibbles & Bits." *Computer Language 7*, 12 (December 1990), 45–55.

[MMIX] Part of a forthcoming edition of *The Art of Computer Programming*. Available at http://www-cs-faculty.stanford.edu/~knuth/taocp.html.

[NZM] Niven, Ivan, Zuckerman, Herbert S., and Montgomery, Hugh L. *An Introduction to the Theory of Numbers*, Fifth Edition. John Wiley & Sons, Inc., 1991.

[PB] Purdom, Paul Walton Jr., and Brown, Cynthia A. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.

[PHO] Oden, Peter H. Private communication.

[PL8] I learned this trick from the PL.8 compiler.

[Rib] Ribenboim, Paulo. *The Little Book of Big Primes*. Springer-Verlag, 1991.

[RND] Reingold, Edward M., Nievergelt, Jurg, and Deo, Narsingh. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[Sagan] Sagan, Hans. *Space-Filling Curves*. Springer-Verlag, 1994. A wonderful book, thoroughly recommended to anyone even slightly interested in the subject.

[Shep] Shepherd, Arvin D. Private communication.

[Stall] Stallman, Richard M. *Using and Porting GNU CC*. Free Software Foundation, 1998.

[Voor] Voorhies, Douglas. "Space-Filling Curves and a Measure of Coherence." *Graphics Gems II*, AP Professional (1991).

[War] Warren, H. S., Jr. "Functions Realizable with Word-Parallel Logical and Two's-Complement Addition Instructions." *Communications of the ACM 20*, 6 (June 1977), 439–441.

[Weg] The earliest reference to this that I know of is: Wegner, P. A. "A Technique for Counting Ones in a Binary Computer." *Communications of the ACM 3*, 5 (May 1960), 322.

[Wells] Wells, David. *The Penguin Dictionary of Curious and Interesting Numbers*. Penguin Books, 1997.

[Will] Willans, C. P. "On Formulae for the *n*th Prime Number." *The Mathematical Gazette 48* (1964), 413–415.

[Wor] Wormell, C. P. "Formulae for Primes." *The Mathematical Gazette 51* (1967), 36–38.

索　引

索引中的页码为英文原书页码，与书中边栏的页码一致。

D

E

F

M

N

O

P

T

U

V-W

Z